CSE1030 – Introduction to Computer Science II

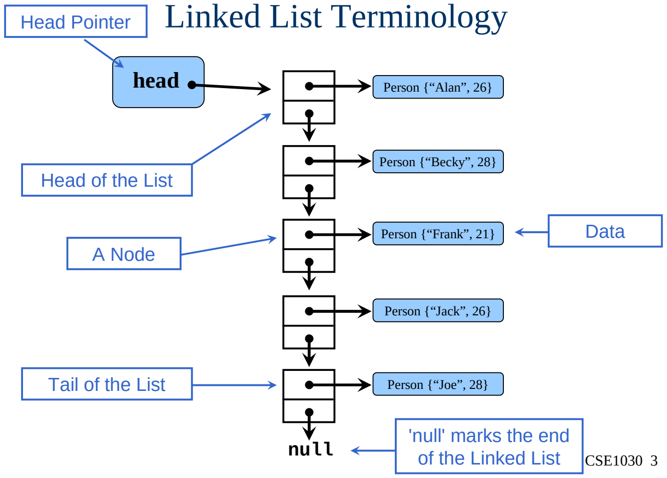
Lecture #18

Linked Lists – Coding Examples

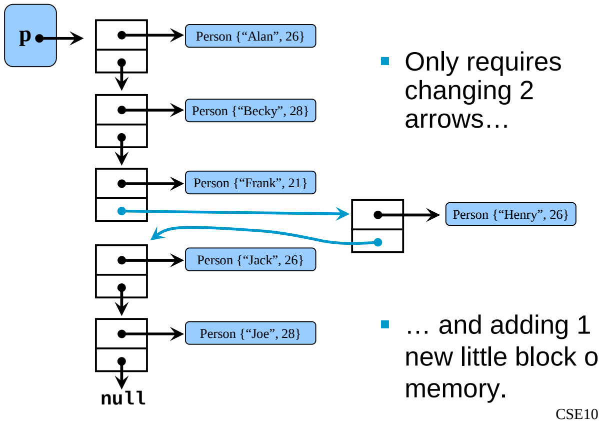
CSE1030 – Lecture #18

- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Linked List Terminology

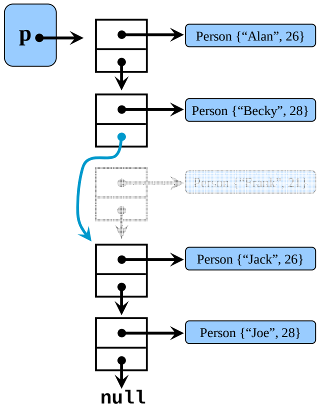


Inserting…



- Only requires changing 2 arrows…
- … and adding 1 new little block of memory.

Deleting…

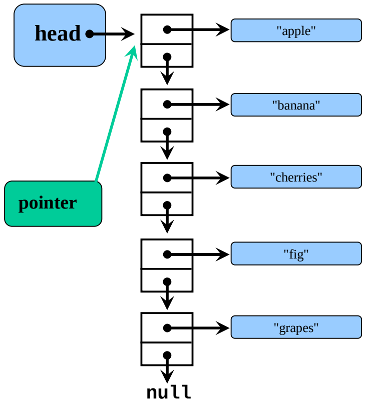


- Deleting "Frank" only requires us to update 1 pointer – Fast!

CSE1030 – Lecture #18

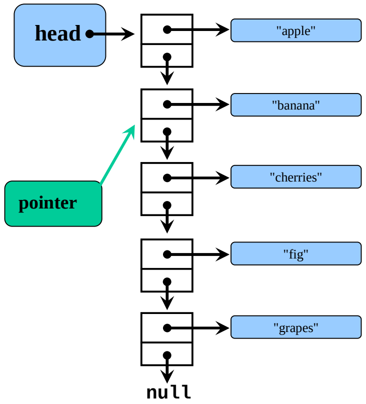
- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Linked List Iteration



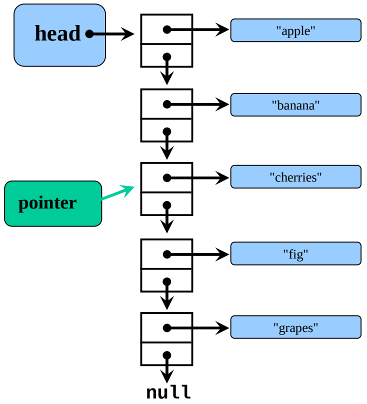
- Iterating through a list means we have to construct a "pointer", and move the pointer along the list, one item at a time.
- We accomplish this by using the "next" pointers

Linked List Iteration



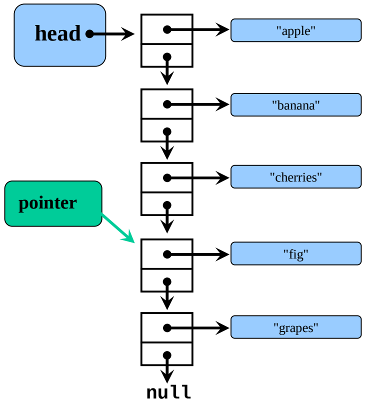
- Iterating through a list means we have to construct a "pointer", and move the pointer along the list, one item at a time.
- We accomplish this by using the "next" pointers

Linked List Iteration



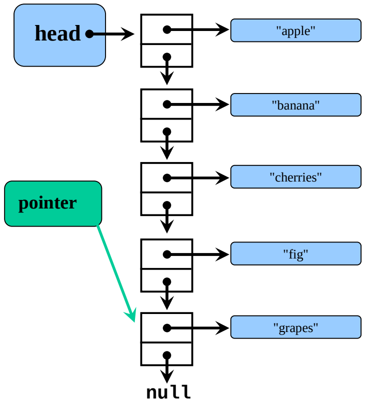
- Iterating through a list means we have to construct a "pointer", and move the pointer along the list, one item at a time.
- We accomplish this by using the "next" pointers

Linked List Iteration



- Iterating through a list means we have to construct a "pointer", and move the pointer along the list, one item at a time.
- We accomplish this by using the "next" pointers

Linked List Iteration



- Iterating through a list means we have to construct a "pointer", and move the pointer along the list, one item at a time.
- We accomplish this by using the "next" pointers

Example Code: iteration

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

This defines the Node class

```
// create a new empty linked-list:
Node head = null;
```

Init the head pointer

```
// insert a node or two:
head = new Node("apple",
          new Node("banana",
             new Node("cherries",
                new Node("fig",
                   new Node("grapes", null)
                )
             )
          )
       );
```

Create the nodes of the linked-list

```
// now we want to output the list:
Node pointer = head;

int i;
while(pointer != null)
{
   System.out.println("   " + i++ + " " + pointer.data);

   pointer = pointer.next;
}

System.out.println("Done!");
```

Start at the head ("top") of the list

Use the Data

Move the pointer on down the list

Output: iteration

```
>java iteration
  0  apple
  1  banana
  2  cherries
  3  fig
  4  grapes
Done!
```

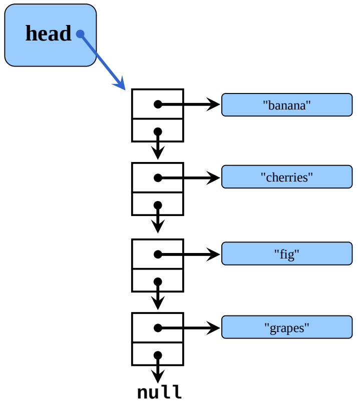
CSE1030 – Lecture #18

- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Inserting Nodes into a Linked-List

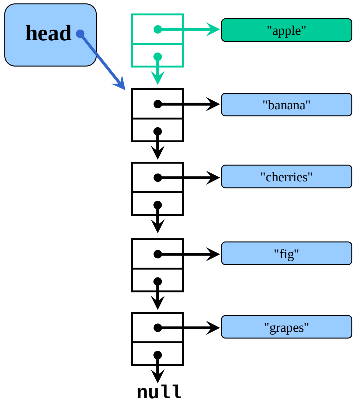
- Insertion requires us to create a new Node, and update a pointer
- There are three cases:
  1. Inserting at the **head** of the list
  2. Inserting at the **end** of the list
  3. Inserting in the **middle**

Inserting at the Beginning



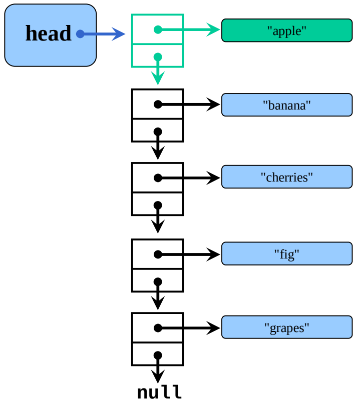
- To insert at the beginning of the list we have to change the head pointer…
- and we have to add a new node that points to the rest of the list.

Inserting at the Beginning



- Here we add the new Node
- Note that the node's 'next' pointer points to where the head pointer currently points (the former top of the list)

Inserting at the Beginning



- Next we update the head pointer and we're done
- Let's look at the code…

Example Code: insertAtHead

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
// create a new empty linked-list:
Node head = null;


void insertAtHead(String data)
{
   // create the new node
   // note that the 'next' pointer for the new
   // node must point to the current Head node
   Node newNode = new Node(data, head);


   // now, update the head pointer
   head = newNode;
}
```

Create the new node, with data and next pointer

Update the Head pointer

```
// create a new empty linked-list:
head = null;

// insert a node or two:
insertAtHead("apple");
insertAtHead("banana");
insertAtHead("cherries");
insertAtHead("fig");
insertAtHead("grapes");


// now we want to output the list:
Node pointer = head;

int i = 0;
while(pointer != null)
{
   System.out.println("  " + i++ + "  " + pointer.data);
   pointer = pointer.next;
}

System.out.println("Done!");
```

New Empty Linked-List

Inserts some Nodes

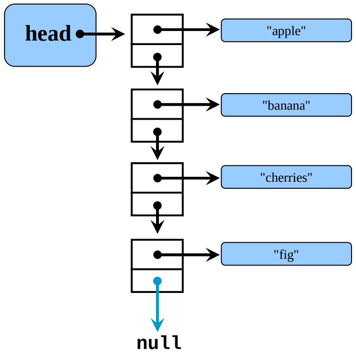
Output the List

Output: insertAtHead

```
>java insertAtHead
  0  grapes
  1  fig
  2  cherries
  3  banana
  4  apple
Done!
```

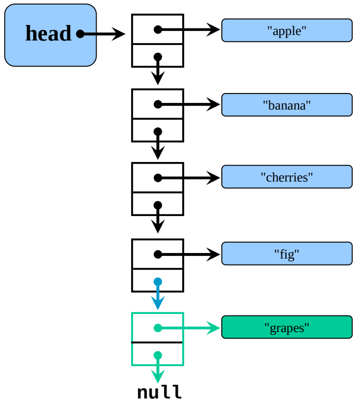
Reverse Alphabetical Order, Why?

Inserting at the End



- To insert at the end of the list we have to change the 'next' pointer of the last node…
- and we have to add a new node with a 'next' pointer that is null.

Inserting at the End



- To insert at the end of the list we have to change the 'next' pointer of the last node…
- and we have to add a new node with a 'next' pointer that is null.
- Let's look at the code

Example Code: insertAtEnd

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
void insertAtEnd(String data)
{
   // create the new node
   // the 'next' pointer will be 'null'
   Node newNode = new Node(data, null);

   // Special Case: if the list is empty
   if(head == null)
   {
      head = newNode;
      return;
   }

   // otherwise, we're looking for the
   // end of the list, which is the node
   // whose 'next' pointer is null
   Node pointer = head;
   while(pointer.next != null)
      pointer = pointer.next;

   // update the tail Node 'next' pointer
   pointer.next = newNode;
}
```

Create the new node, with data and next pointer

Handle Empty List

Find the Tail Node of the List

Update 'next' pointer in the Tail Node

```
 // create a new empty linked-list:
head = null;

// insert a node or two:
insertAtEnd("apple");
insertAtEnd("banana");
insertAtEnd("cherries");
insertAtEnd("fig");
insertAtEnd("grapes");

// now we want to output the list:
Node pointer = head;

int i = 0;
while(pointer != null)
{
   System.out.println("  " + i++ + "   " + pointer.data);
   pointer = pointer.next;
}

System.out.println("Done!");
```

New Empty Linked-List

Inserts some Nodes

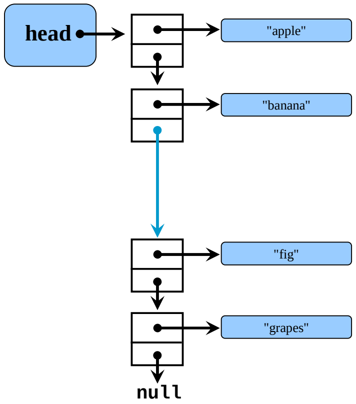
Output the List

Output: insertAtEnd

```
>java insertAtEnd
  0  apple
  1  banana
  2  cherries
  3  fig
  4  grapes
Done!
```

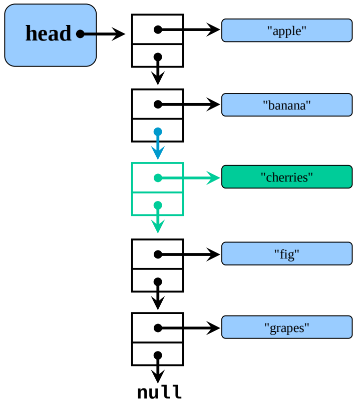
This time in Alphabetical Order, Why?

Inserting in the Middle

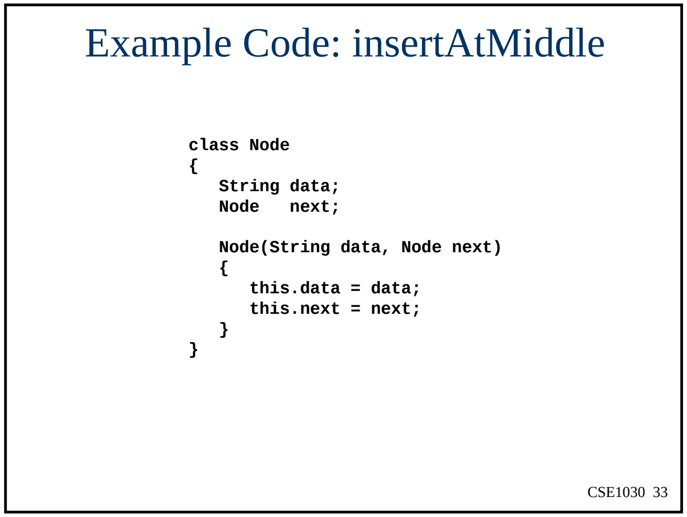


- To insert in the middle of the list we have to find the node **above** where the new node should go…
- because that's the node where the 'next' pointer has to be changed.
- Let's look at the code…

Inserting in the Middle

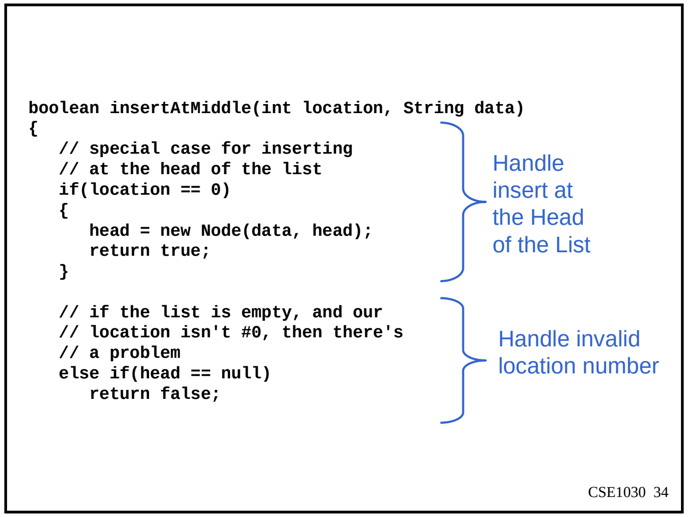


- To insert in the middle of the list we have to find the node **above** where the new node should go…
- because that's the node where the 'next' pointer has to be changed.



Example Code: insertAtMiddle

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

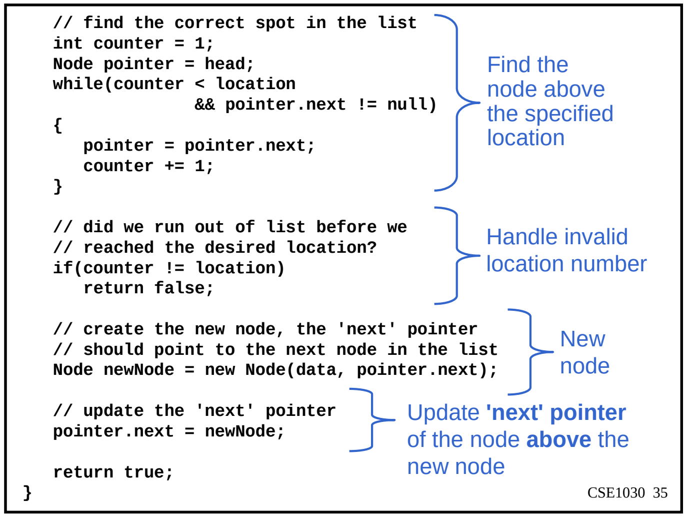


```
boolean insertAtMiddle(int location, String data)
{
   // special case for inserting
   // at the head of the list
   if(location == 0)
   {
      head = new Node(data, head);
      return true;
   }
```

Handle insert at the Head of the List

```
   // if the list is empty, and our
   // location isn't #0, then there's
   // a problem
   else if(head == null)
      return false;
```

Handle invalid location number



```
   // find the correct spot in the list
   int counter = 1;
   Node pointer = head;
   while(counter < location
                 && pointer.next != null)
   {
      pointer = pointer.next;
      counter += 1;
   }
```

Find the node above the specified location

```
   // did we run out of list before we
   // reached the desired location?
   if(counter != location)
      return false;
```

Handle invalid location number

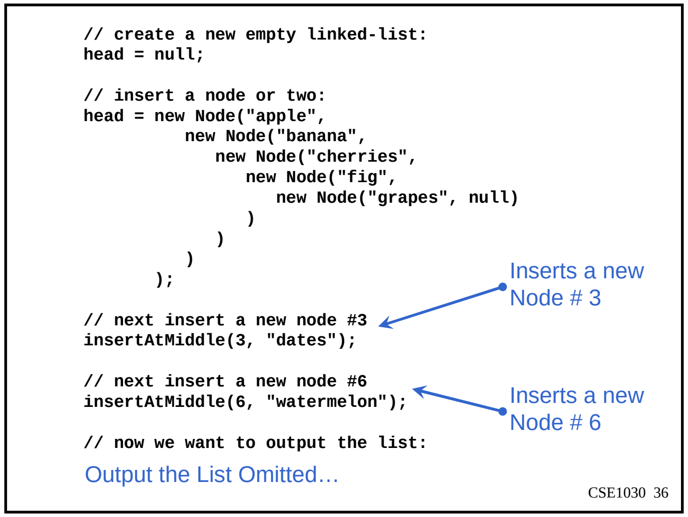
```
   // create the new node, the 'next' pointer
   // should point to the next node in the list
   Node newNode = new Node(data, pointer.next);
```

New node

```
   // update the 'next' pointer
   pointer.next = newNode;
```

Update **'next' pointer** of the node **above** the new node

```
   return true;
}
```



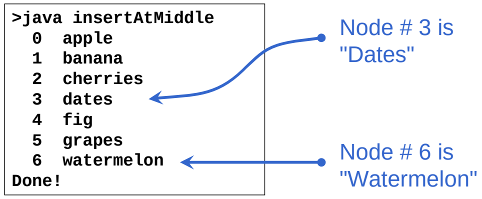
```
// create a new empty linked-list:
head = null;

// insert a node or two:
head = new Node("apple",
          new Node("banana",
             new Node("cherries",
                new Node("fig",
                   new Node("grapes", null)
                )
             )
          )
       );

// next insert a new node #3
insertAtMiddle(3, "dates");

// next insert a new node #6
insertAtMiddle(6, "watermelon");

// now we want to output the list:
```

Inserts a new Node # 3

Inserts a new Node # 6

Output the List Omitted…

Output: insertAtMiddle



```
>java insertAtMiddle
  0  apple
  1  banana
  2  cherries
  3  dates
  4  fig
  5  grapes
  6  watermelon
Done!
```

Node # 3 is "Dates"

Node # 6 is "Watermelon"

Another Example: insertInOrder

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
void insertInOrder(String data)
{
   // special case for inserting into
   // an empty list
   if(head == null)
   {
      head = new Node(data, null);
      return;
   }
```

Handle insert into empty List

```
   // do we come before the first element?
   // then we have to update the head
   // pointer
   if(head.data.compareTo(data) > 0)
   {
      head = new Node(data, head);
      return;
   }
```

Handle insert at the Head of the List

```
   // find the correct spot in the list
   Node pointer = head;
   while(pointer.next != null
      && pointer.next.data.compareTo(data) < 0)
      pointer = pointer.next;
```

Find the node above the correct spot

```
   // create the new node, the 'next' pointer should
   // point to the next node in the list
   Node newNode = new Node(data, pointer.next);
```

New node

```
   // update the 'next' pointer
   // of the previous node
   pointer.next = newNode;
}
```

Update **'next' pointer** of the node **above** the new node

```
// create a new empty linked-list:
head = null;

// insert a node or two:
insertInOrder("cherries");
insertInOrder("watermelon");
insertInOrder("fig");
insertInOrder("banana");
insertInOrder("dates");
insertInOrder("apple");
insertInOrder("grapes");


// now we want to output the list:
Node pointer = head;

int i = 0;
while(pointer != null)
{
   System.out.println("  " + i++ + "  " + pointer.data);
   pointer = pointer.next;
}

System.out.println("Done!");
```

Here we're inserting the nodes into an empty linked-list in a 'random' order

Output: insertInOrder

```
>java insertInOrder
  0  apple
  1  banana
  2  cherries
  3  dates
  4  fig
  5  grapes
  6  watermelon
Done!
```

The Nodes are in alphabetical order, although they were not inserted in that order

CSE1030 – Lecture #18

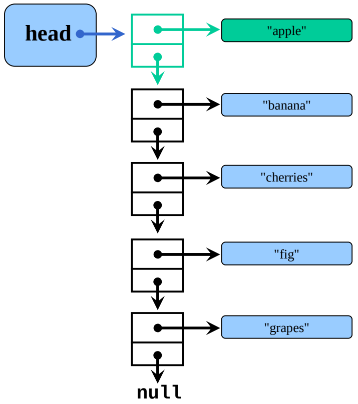
- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Deleting Nodes from a Linked-List

- Deletion only requires us to update a pointer

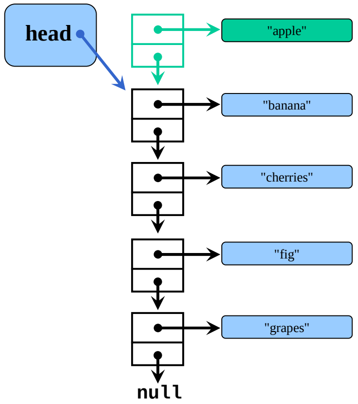
- There are three cases:
  1. Deleting from the **head** of the list
  2. Deleting from the **end** of the list
  3. Deleting from the **middle**

Deleting from the Beginning



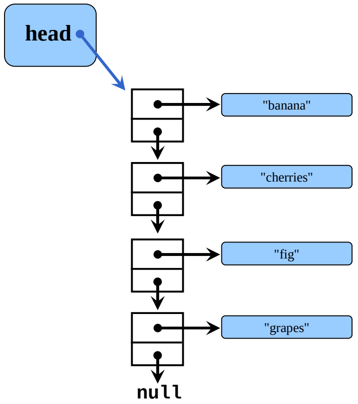
- All we have to do is change the head pointer
- Java's "Garbage Collection" will figure-out that there is no longer a pointer to the first node, and destroy it

Deleting from the Beginning



- Here we move the head pointer one node down the list…

Deleting from the Beginning



- … and now the old head node is gone
- let's look at the code…

Example Code: deleteFromHead

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
Node deleteFromHead()
{
   // if the list is empty, then there's
   // nothing to do
   if(head == null)
      return null;


   // remember the old head node, so
   // we can return it (just in case
   // the user wants it)
   Node oldHead = head;


   // now, update the head pointer
   head = head.next;


   return oldHead;
}
```

Handle Empty List

Grab a reference to the node we're deleting, to return

Here's where we delete the node

Return a reference to the deleted node

```
// create a new empty linked-list:
head = null;

// insert a node or two:
head = new Node("apple",
          new Node("banana",
             new Node("cherries",
                new Node("fig",
                   new Node("grapes", null)
                )
             )
          )
       );

// now delete a node...
Node first = deleteFromHead();
System.out.println("Deleted: " + first.data);

// now delete a node...
Node second = deleteFromHead();
System.out.println("Deleted: " + second.data);

// now we want to output the list:
```

Delete "apple"

Delete "banana"

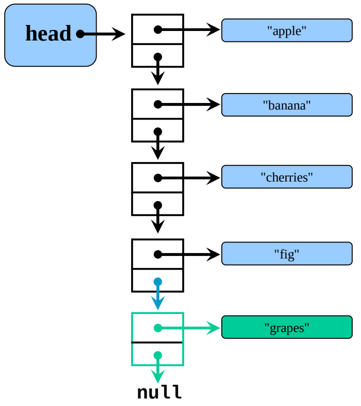
Output: deleteFromHead

```
>java deleteFromHead
Deleted: apple
Deleted: banana
  0  cherries
  1  fig
  2  grapes
Done!
```

Two deleted nodes …

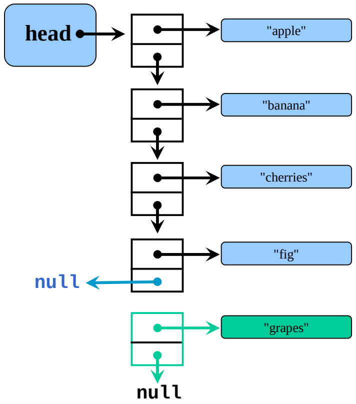
…are not in the list anymore

Deleting from the End



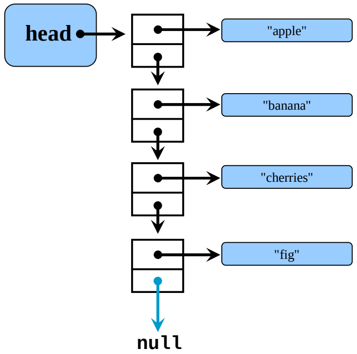
- To delete from the end of the list we have to change the 'next' pointer of the **second-last** node…

Deleting from the End



- To delete from the end of the list we have to change the 'next' pointer of the **second-last** node to null…

Deleting from the End



- To delete from the end of the list we have to change the 'next' pointer of the **second-last** node to null…
- Let's look at the code

Example Code: deleteFromEnd

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
Node deleteFromEnd()
{
   // if the list is empty, then there's
   // nothing to do
   if(head == null)
      return null;
```

Handle Empty List

```
   // if the list only has 1 node
   // delete from head of list
   if(head.next == null)
   {
      Node oldHead = head;
      head = null;
      return oldHead;
   }
```

If there is only 1 node, then this is the same as "delete from head of the list"

```
   // otherwise, we're looking for the
   // second-last node of the list, which
   // is the node whose '.next.next'
   // pointer is null
   Node pointer = head;
   while(pointer.next.next != null)
      pointer = pointer.next;
```

Find the second from last node

```
   // remember the deleted node, so
   // we can return it (just in case
   // the user wants it)
   Node deletedNode = pointer.next;
```

Grab a reference to the node we're deleting, to return

```
   // now, update the 'next' pointer
   pointer.next = null;
```

Here's where we delete the node

```
   return deletedNode;
}
```

Return a reference to the deleted node

```
// create a new empty linked-list:
head = null;

// insert a node or two:
head = new Node("apple",
          new Node("banana",
             new Node("cherries",
                new Node("fig",
                   new Node("grapes", null)
                )
             )
          )
       );

// now delete a node...
Node first = deleteFromEnd();
System.out.println("Deleted: " + first.data);

// now delete a node...
Node second = deleteFromEnd();
System.out.println("Deleted: " + second.data);

// now we want to output the list:
```

Delete "grapes"

Delete "fig"

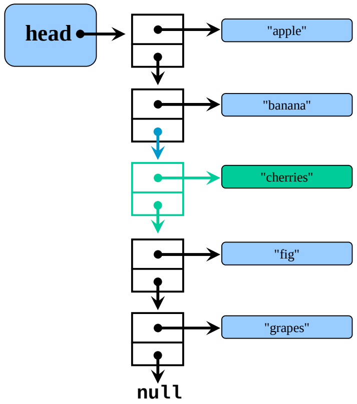
Output: deleteFromEnd

```
>java deleteFromEnd
Deleted: grapes
Deleted: fig
  0  apple
  1  banana
  2  cherries
Done!
```

Two deleted nodes …

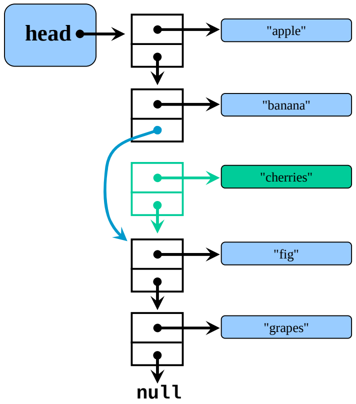
…are not in the list anymore

Deleting from the Middle



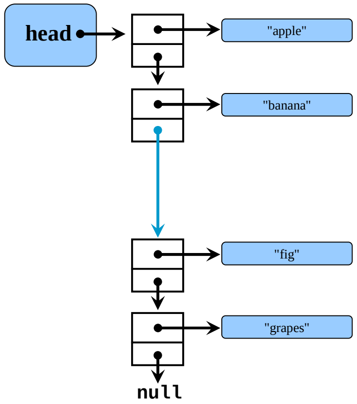
- To delete from the middle of the list we have to find the node **above** the node we want to delete…
- because that's the node where the 'next' pointer has to be changed.

Deleting from the Middle



- Once we have updated the preceding 'next' pointer, the skipped node has been removed from the list

Deleting from the Middle



- Let's look at the code…

Example Code: deleteFromMiddle

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
static Node deleteFromMiddle(int location)
{
   // empty list?
   if(head == null)
      return null;



   // location is the top of the list
   if(location == 0)
   {
      Node oldHead = head;
      head = head.next;
      return oldHead;
   }
```

Handle Empty List

Deleting from location #0 is the same as "delete from head of the list"

```
// otherwise, we're looking for the
// node above the one we want to delete
int counter = 1;
Node pointer = head;
while(counter < location
   && pointer.next != null
   && pointer.next.next != null)
{
   pointer = pointer.next;
   counter += 1;
}
```

Find the node above the one we want to delete

Why is this here?

```
// did we run out of list before we
// reached the desired location?
if(counter != location)
   return null;
```

Handle invalid location number

```
   // remember the deleted node, so
   // we can return it
   Node deletedNode = pointer.next;
```

Grab a reference to the node we're deleting, to return

```
   // now, update the 'next' pointer
   pointer.next = pointer.next.next;
```

Here's where we delete the node

```
   return deletedNode;
}
```

Return a reference to the deleted node

```
// create a new empty linked-list:
head = null;

// insert a node or two:
head = new Node("apple",
          new Node("banana",
             new Node("cherries",
                new Node("fig",
                   new Node("grapes", null)
                )
             )
          )
       );

// now delete a node...
Node first = deleteFromMiddle(2);
System.out.println("Deleted: " + first.data);

// now delete a node...
Node second = deleteFromMiddle(3);
System.out.println("Deleted: " + second.data);

// now we want to output the list:
```

Delete "cherries"

Delete "grapes"

Output: deleteFromMiddle

```
>java deleteFromMiddle
Deleted: cherries
Deleted: grapes
  0  apple
  1  banana
  2  fig
Done!
```

Two deleted nodes …

…are not in the list anymore

Another Example: deleteByValue

```
class Node
{
   String data;
   Node   next;

   Node(String data, Node next)
   {
      this.data = data;
      this.next = next;
   }
}
```

```
static Node deleteByValue(String data)
{
   // empty list?
   if(head == null)
      return null;
```

Handle Empty List

```
   // location is the top of the list
   if(head.data.equals(data))
   {
      Node oldHead = head;
      head = head.next;
      return oldHead;
   }
```

Check whether we're deleting from the head of the list, and handle it

```
// otherwise, we're looking for the
// node above the one we want to delete
Node pointer = head;
while(pointer.next != null
   && !pointer.next.data.equals(data))
{
   pointer = pointer.next;
}
```

Find the node above the one we want to delete

```
// not found?
if(pointer.next == null)
   return null;
```

The data to be deleted is not found

```
   // remember the deleted node, so
   // we can return it
   Node deletedNode = pointer.next;
```

Grab a reference to the node we're deleting, to return

```
   // now, update the 'next' pointer
   pointer.next = pointer.next.next;
```

Here's where we delete the node

```
   return deletedNode;
}
```

Return a reference to the deleted node

```
// create a new empty linked-list:
head = null;

// insert a node or two:
head = new Node("apple",
          new Node("banana",
             new Node("cherries",
                new Node("fig",
                   new Node("grapes", null)
                )
             )
          )
       );

// now delete a node...
Node first = deleteByValue("banana");
System.out.println("Deleted: " + first.data);

// now delete a node...
Node second = deleteByValue("fig");
System.out.println("Deleted: " + second.data);

// now we want to output the list:
```

Delete "banana"

Delete "fig"

Output: deleteByValue

```
>java deleteByValue
Deleted: banana
Deleted: fig
  0  apple
  1  cherries
  2  grapes
Done!
```

Two deleted nodes …

…are not in the list anymore

Summary of Singly Linked-List Operations

- We have to check whether the operation involves the top of the list, and handle that case separately, because those operations involve changing the **head** pointer
- Otherwise, we must find the node that is 'above' the one we are interested in, because that's the node whose 'next' pointer we have to adjust

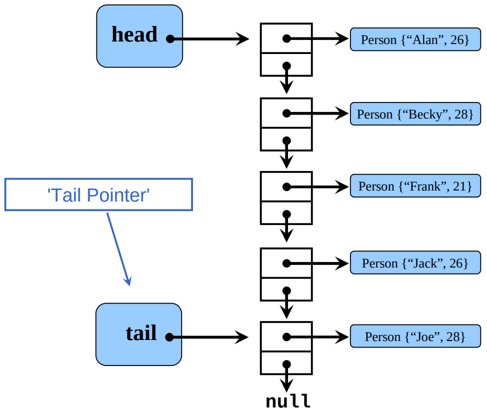
CSE1030 – Lecture #18

- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Common Singly Linked-List Enhancements

- There are two common variations on singly linked-lists:
  - Tail pointers
  - Circular Linked-Lists

Linked List with Tail Pointer

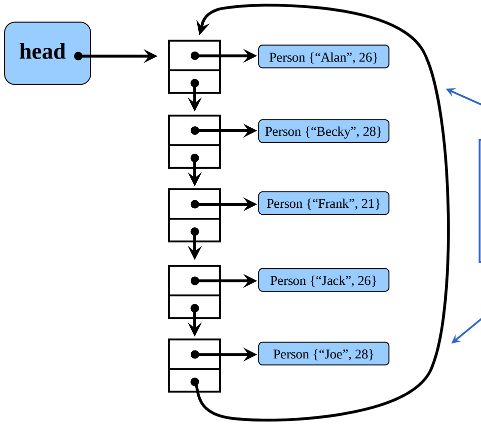


'Tail Pointer'

Tail Pointers

- A Tail Pointer allows the implementer to access the tail quickly
- This makes it fast to add nodes to the tail of the linked list
  - By adding to the tail, and removing from the head, the nodes stay in the order that they were inserted, instead of being reversed.
- As nodes are added or deleted, the Tail Pointer may have to be updated too

Circular Linked Lists



The 'next' pointer of the last node loops back to the top

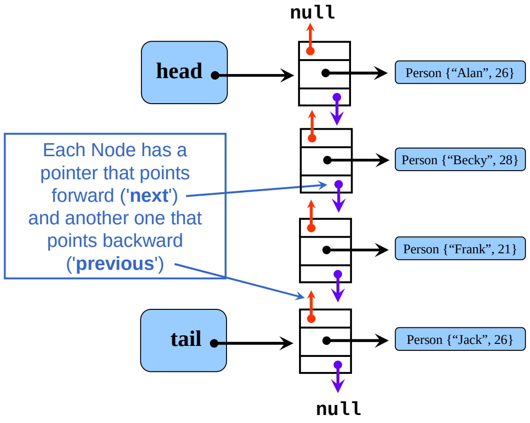
Circular Linked-Lists

- Circular Linked-Lists are useful if the data naturally form a loop, for example:
  - The vertexes around a polygon
  - Algorithms that provide 'fairness' in the allocation to resources
  - 'Round-Robin' waiting lists
  - List of Directions of objects around you

CSE1030 – Lecture #18

- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

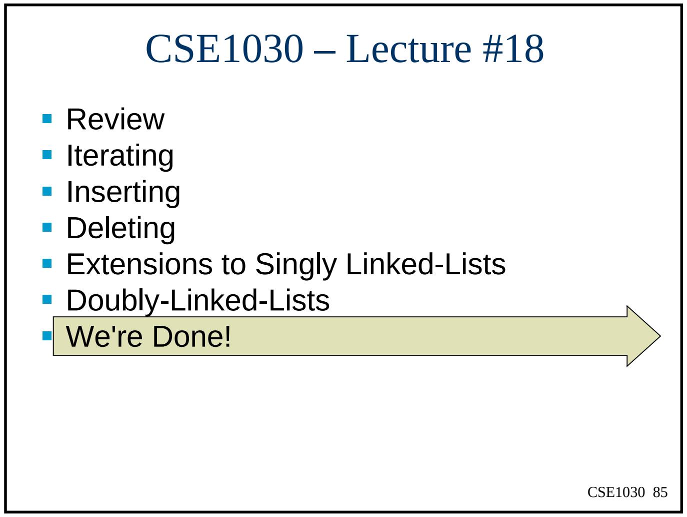
Doubly Linked-List



Each Node has a pointer that points forward ('**next**') and another one that points backward ('**previous**')

Doubly Linked-Lists

- Advantages:
  - Fast to add nodes at either end
  - Easy to iterate in either direction
  - And it is fast to insert a node into the middle (we don’t need to iterate to find the previous node – we already have a pointer directly to it)
- But…
  - And more memory space may be required to store the extra 'previous' pointers
- The code is a little more complicated because more node pointers need to be modified to insert or delete a node



CSE1030 – Lecture #18

- Review
- Iterating
- Inserting
- Deleting
- Extensions to Singly Linked-Lists
- Doubly-Linked-Lists
- We're Done!

Next topic…

Recursion I