

CSE1030 – Introduction to Computer Science II

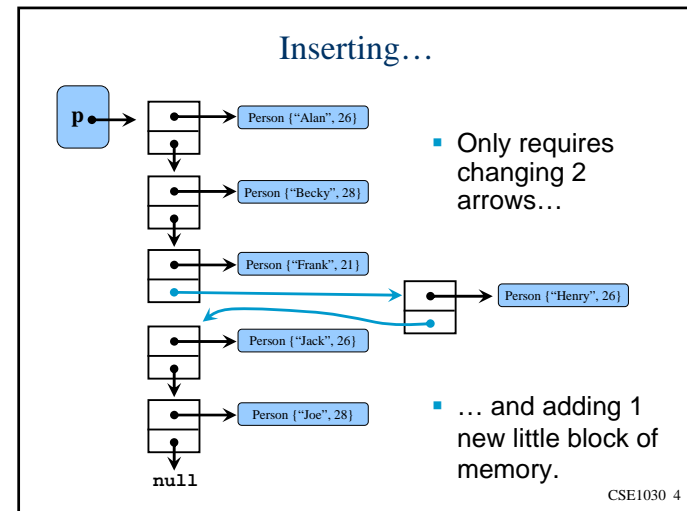
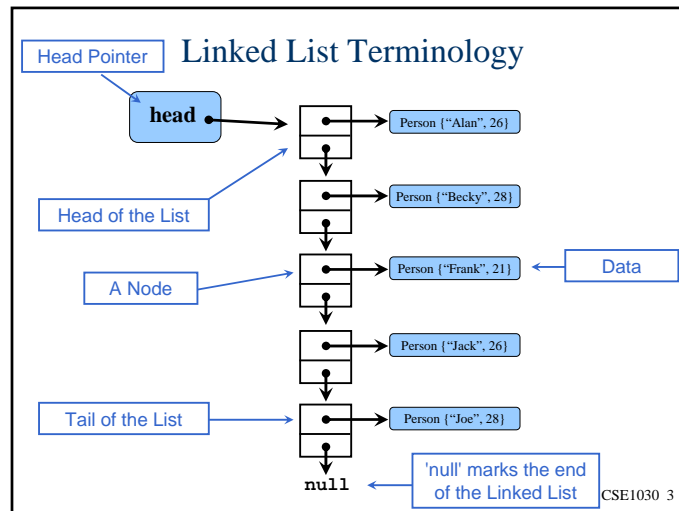
Lecture #18

Linked Lists – Coding Examples

CSE1030 – Lecture #18

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

CSE1030 2



Deleting...

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

CSE1030 5

CSE1030 – Lecture #18

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

CSE1030 6

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

CSE1030 7

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

CSE1030 8

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

CSE1030 9

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

CSE1030 10

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

CSE1030 11

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

CSE1030 12

```
// 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

CSE1030 13

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

```
Node pointer = head;
```

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

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

Use the Data

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

Move the pointer
on down the list

CSE1030 14

Output: iteration

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

CSE1030 15

CSE1030 – Lecture #18

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

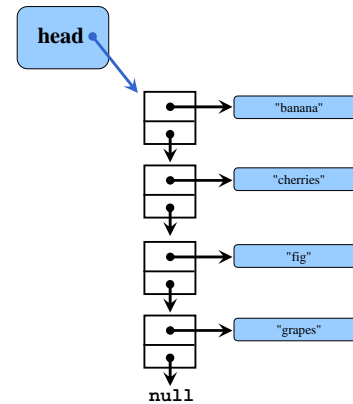
CSE1030 16

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**

CSE1030 17

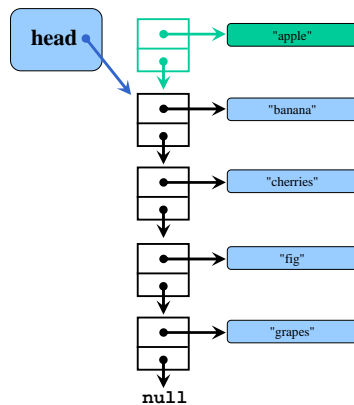
Inserting at the Beginning



- 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.

CSE1030 18

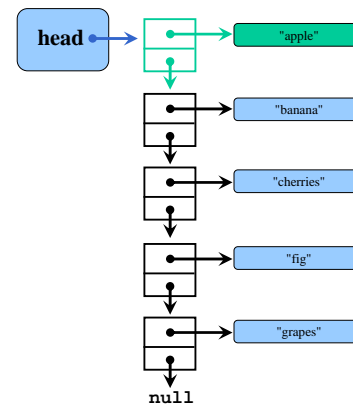
Inserting at the Beginning



- 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)

CSE1030 19

Inserting at the Beginning



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

CSE1030 20

Example Code: insertAtHead

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

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

CSE1030 21

```
// 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);
```

Create the new node, with data and next pointer

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

Update the Head pointer

CSE1030 22

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

New Empty Linked-List

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

Inserts some Nodes

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

Output the List

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

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

CSE1030 23

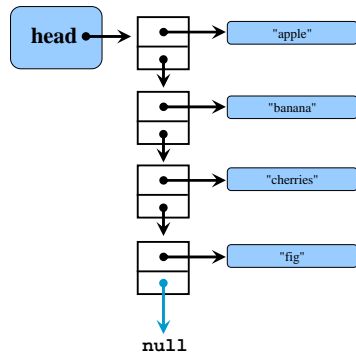
Output: insertAtHead

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

Reverse Alphabetical Order, Why?

CSE1030 24

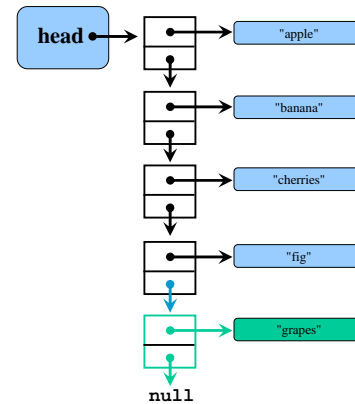
Inserting at the End



- 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.

CSE1030 25

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

CSE1030 26

Example Code: insertAtEnd

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

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

CSE1030 27

```
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

CSE1030 28

```

// 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

CSE1030 29

Output: insertAtEnd

```

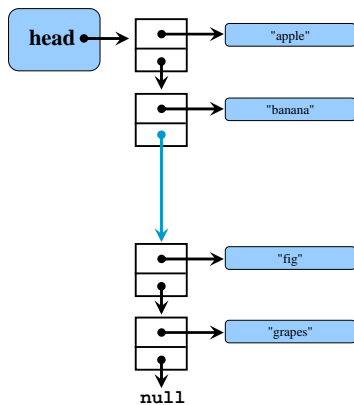
>java insertAtEnd
0 apple
1 banana
2 cherries
3 fig
4 grapes
Done!

```

This time in
Alphabetical
Order,
Why?

CSE1030 30

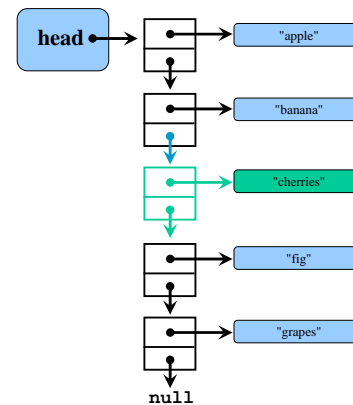
Inserting in the Middle



- 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...

CSE1030 31

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.

CSE1030 32

Example Code: insertAtMiddle

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

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

CSE1030 33

```
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;
    }

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

Handle insert at the Head of the List

Handle invalid location number

CSE1030 34

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

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

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

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

return true;
}
```

Find the node above the specified location

Handle invalid location number

New node

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

CSE1030 35

```
// 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:
Output the List Omitted...
```

Inserts a new Node # 3

Inserts a new Node # 6

CSE1030 36

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"

CSE1030 37

Another Example: insertInOrder

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

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

CSE1030 38

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

    // 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 into empty List

Handle insert at the Head of the List

CSE1030 39

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

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

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

Find the node above the correct spot

New node

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

CSE1030 40

```
// 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");
```

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

```
// 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!");
```

CSE1030 41

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 42

CSE1030 – Lecture #18

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

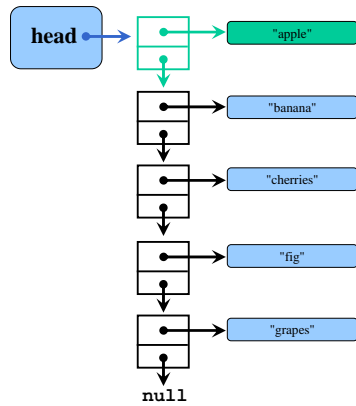
CSE1030 43

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**

CSE1030 44

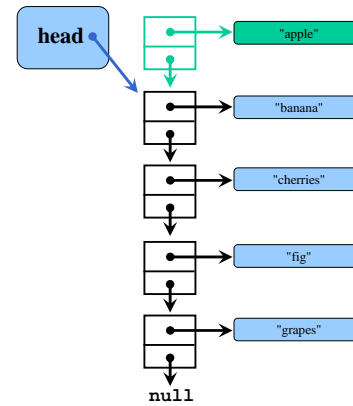
Deleting from the Beginning



- 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

CSE1030 45

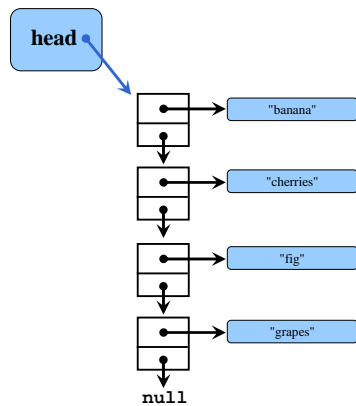
Deleting from the Beginning



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

CSE1030 46

Deleting from the Beginning



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

CSE1030 47

Example Code: deleteFromHead

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

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

CSE1030 48

```

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

CSE1030 49

```

// 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"

CSE1030 50

Output: deleteFromHead

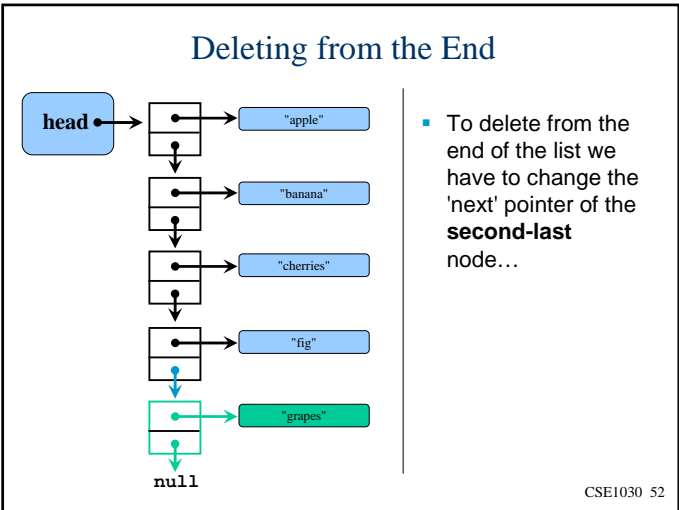
```

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

```

Two deleted nodes ...
 ...are not in the list anymore

CSE1030 51



Deleting from the End

head → [apple] → [banana] → [cherries] → [fig] → [grapes] → null

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

CSE1030 53

Deleting from the End

head → [apple] → [banana] → [cherries] → [fig] → null

- 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

CSE1030 54

Example Code: deleteFromEnd

```

class Node
{
    String data;
    Node next;

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

```

CSE1030 55

```

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

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

```

Handle Empty List

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

CSE1030 56

```

// 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;

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

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

return deletedNode;
}

```

Find the second from last node

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

CSE1030 57

```

// 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"

CSE1030 58

Output: deleteFromEnd

```

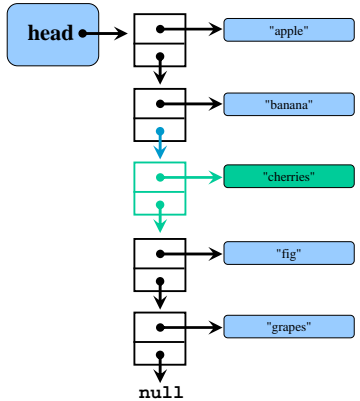
>java deleteFromEnd
Deleted: grapes
Deleted: fig
0 apple
1 banana
2 cherries
Done!

```

Two deleted nodes ...

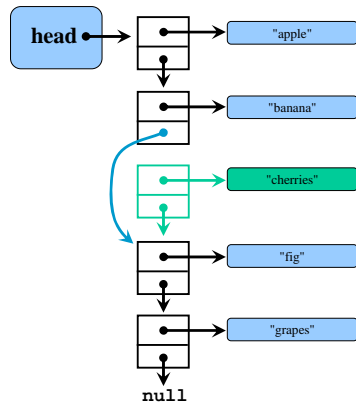
...are not in the list anymore

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.

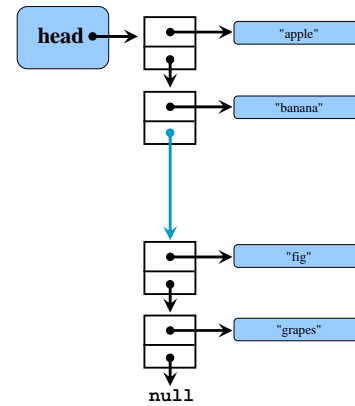
Deleting from the Middle



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

CSE1030 61

Deleting from the Middle



- Let's look at the code...

CSE1030 62

Example Code: deleteFromMiddle

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

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

CSE1030 63

```
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 = null;
        return oldHead;
    }
}
```

} Handle
Empty
List

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

CSE1030 64


```

// 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

CSE1030 65

```

// 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

CSE1030 66

```

// 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)
      )
    )
  )
);

```

Delete "cherries"

```

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

```

Delete "grapes"

```

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

```

```

// now we want to output the list:

```

CSE1030 67

Output: deleteFromMiddle

```

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

```

Two deleted nodes ...

...are not in the list anymore

CSE1030 68

Another Example: deleteByValue

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

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

CSE1030 69

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

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

Handle Empty List

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

CSE1030 70

```
// 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

CSE1030 71

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

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

return deletedNode;
```

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

CSE1030 72

```

// 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"

CSE1030 73

Output: deleteByValue

```

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

```

Two deleted
nodes ...

...are not in the
list anymore

CSE1030 74

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 75

CSE1030 – Lecture #18

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

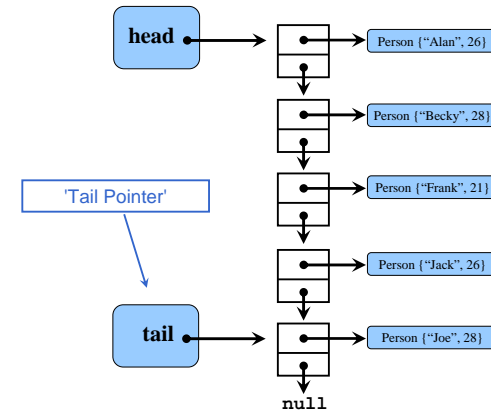
CSE1030 76

Common Singly Linked-List Enhancements

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

CSE1030 77

Linked List with Tail Pointer



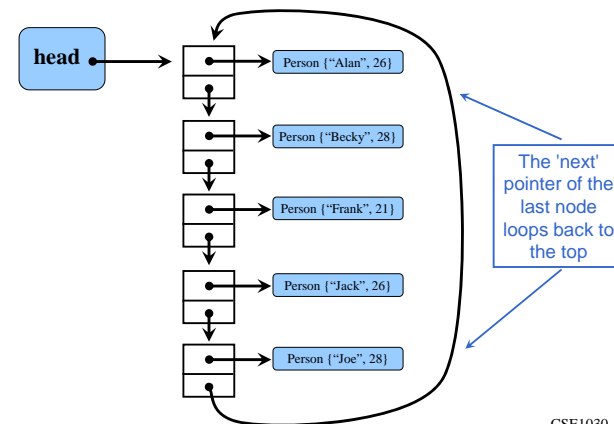
CSE1030 78

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

CSE1030 79

Circular Linked Lists



CSE1030 80

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

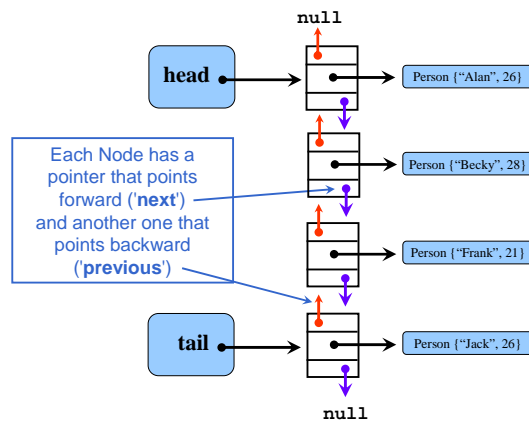
CSE1030 81

CSE1030 – Lecture #18

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

CSE1030 82

Doubly Linked-List



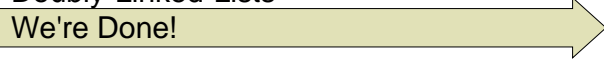
CSE1030 83

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 84

CSE1030 – Lecture #18

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

CSE1030 85

Next topic...

Recursion I

CSE1030 86