

**CSC 124**  
<http://faculty.cua.edu/elsharkawy/csc-124.htm>

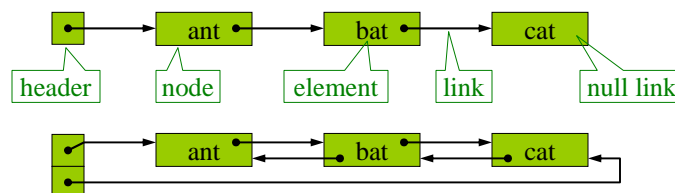
# Abstract Data Types: Linked Lists

Instructor  
**Sameh Elsharkawy, Ph.D.**  
[Elsharkawy@cua.edu](mailto:Elsharkawy@cua.edu)



## Linked lists (1)

- A linked list consists of a sequence of nodes connected by links, plus a header.
- Each node (except the last) has a successor, and each node (except the first) has a predecessor.
- Each node contains a single element (object or value), plus links to its successor and/or predecessor.



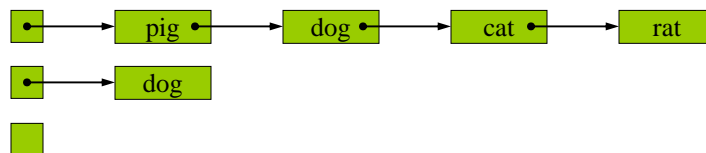
## Linked lists (2)

- **The length of a linked list is the number of nodes.**
- **An empty linked list has no nodes.**
- **In a linked list:**
  - We can manipulate the individual elements.
  - We can manipulate the links, thus changing the linked list's very structure! (This is impossible in an array.)

3

## Singly-linked lists (1)

- **A singly-linked list (SLL) consists of a sequence of nodes, connected by links in one direction only.**
- **Each SLL node contains a single element, plus a link to the node's successor (or a null link if the node has no successor).**
- **An SLL header contains a link to the SLL's first node (or a null link if the SLL is empty).**



4

## Singly-linked lists (2)

- Java class implementing SLL nodes:

```
public class SLLNode {
    protected Object element;
    protected SLLNode succ;
    public SLLNode (Object elem, SLLNode succ) {
        this.element = elem;
        this.succ = succ;
    }
}
```

5

## Singly-linked lists (3)

- Java class implementing SLL headers:

```
public class SLL {
    private SLLNode first;
    public SLL () {
        // Construct an empty SLL.
        this.first = null;
    }

    ...

}
```

SLL methods (to follow)

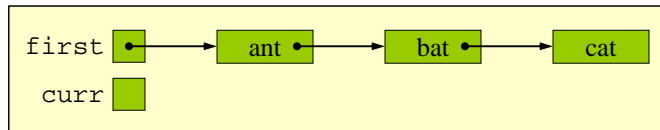
6

## Example: SLL traversal

- Instance method (in class `SLL`) to traverse an SLL:

```
public void printFirstToLast () {
    // Print all elements in this SLL, in first-to-last order.
    for (SLLNode curr = this.first;
        curr != null; curr = curr.succ)
        System.out.println(curr.element);
}
```

- Animation:



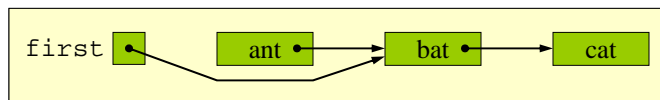
7

## Example: SLL manipulation (1)

- Instance method (in class `SLL`) to delete an SLL's first node:

```
public void deleteFirst () {
    // Delete this SLL's first node (assuming length > 0).
    this.first = this.first.succ;
}
```

- Animation:



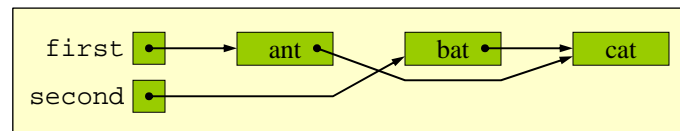
8

## Example: SLL manipulation (2)

- Instance method (in class `SLL`) to delete an SLL's second node:

```
public void deleteSecond () {
    // Delete this SLL's second node (assuming length > 1).
    SLLNode second = this.first.succ;
    this.first.succ = second.succ;
}
```

- Animation:



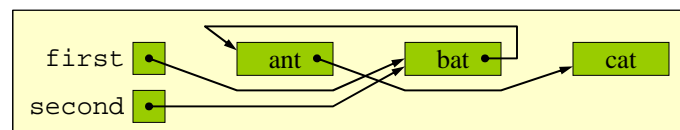
9

## Example: SLL manipulation (3)

- Instance method (in class `SLL`) to swap an SLL's first and second nodes:

```
public void swapFirstTwo () {
    // Swap this SLL's 1st and 2nd nodes (assuming length > 1).
    SLLNode second = this.first.succ;
    this.first.succ = second.succ;
    second.succ = this.first;
    this.first = second;
}
```

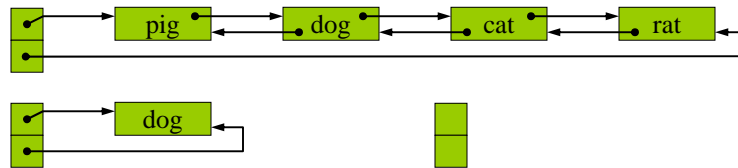
- Animation:



10

## Doubly-linked lists (1)

- A doubly-linked list (DLL) consists of a sequence of nodes, connected by links in both directions.
- Each DLL node contains a single element, plus links to the node's successor and predecessor (or null link(s)).
- The DLL header contains links to the DLL's first and last nodes (or null links if the DLL is empty).



11

## Doubly-linked lists (2)

- Java class implementing DLL nodes:

```
public class DLLNode {
    protected Object element;
    protected DLLNode pred, succ;
    public DLLNode (Object elem,
                    DLLNode pred, DLLNode succ) {
        this.element = elem;
        this.pred = pred; this.succ = succ;
    }
}
```

12

## Doubly-linked lists (3)

- **Java class implementing DLL headers:**

```
public class DLL {
    private DLLNode first, last;
    public DLL () {
        // Construct an empty DLL.
        this.first = null;
        this.last = null;
    }

    ...

}
```

DLL methods (to follow)

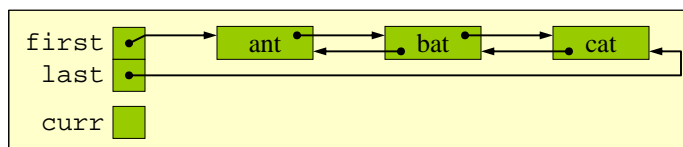
13

## Example: DLL traversal

- **Instance method (in class DLL) to traverse a DLL, from last node to first:**

```
public void printLastToFirst () {
    // Print all elements in this DLL, in last-to-first order.
    for (DLLNode curr = this.last;
         curr != null; curr = curr.pred)
        System.out.println(curr.element);
}
```

- **Animation:**



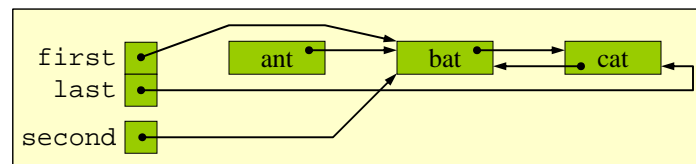
14

## Example: DLL manipulation (1)

- Instance method (in class `DLL`) to delete a `DLL`'s first node:

```
public void deleteFirst () {
    // Delete this DLL's first node (assuming length > 0).
    DLLNode second = this.first.succ;
    second.pred = null;
    this.first = second;
}
```

- Animation:



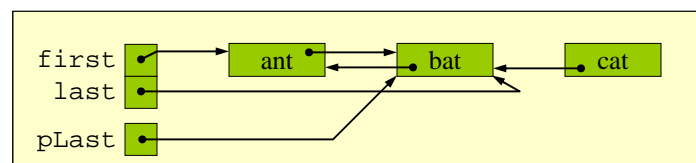
15

## Example: DLL manipulation (2)

- Instance method (in class `DLL`) to delete a `DLL`'s last node:

```
public void deleteLast () {
    // Delete this DLL's last node (assuming length > 0).
    DLLNode pLast = this.last.pred;
    pLast.succ = null;
    this.last = pLast;
}
```

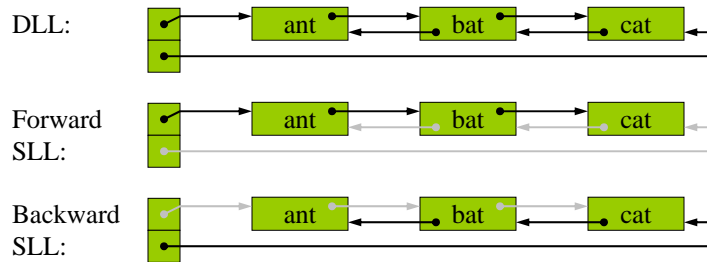
- Animation:



16

## DLL = forward SLL + backward SLL

- View a DLL as a backward SLL superimposed on a forward SLL:



17

## Insertion

- Problem:** Insert a new element at a given point in a linked list.
- Four cases to consider:**
  - insertion in an empty linked list;
  - insertion before the first node of a nonempty linked list;
  - insertion after the last node of a nonempty linked list;
  - insertion between nodes of a nonempty linked list.
- The insertion algorithm needs links to the new node's successor and predecessor.**

18

## SLL insertion (1)

- **SLL insertion algorithm:**

To insert *elem* at a given point in the SLL headed by *first*:

1. Make *ins* a link to a newly-created node with element *elem* and successor null.
2. If the insertion point is before the first node:
  - 2.1. Set node *ins*'s successor to *first*.
  - 2.2. Set *first* to *ins*.
3. If the insertion point is after the node *pred*:
  - 3.1. Set node *ins*'s successor to node *pred*'s successor.
  - 3.2. Set node *pred*'s successor to *ins*.
4. Terminate.

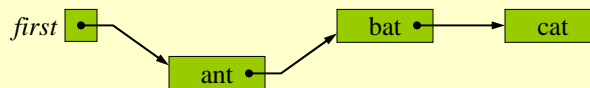
19

## SLL insertion (2)

- **Animation (insertion before first node):**

To insert *elem* at a given point in the SLL headed by *first*:

1. Make *ins* a link to a newly-created node with element *elem* and successor null.
2. If the insertion point is before the first node:
  - 2.1. Set node *ins*'s successor to *first*.
  - 2.2. Set *first* to *ins*.
3. If the insertion point is after the node *pred*:
  - 3.1. Set node *ins*'s successor to node *pred*'s successor.
  - 3.2. Set node *pred*'s successor to *ins*.
4. **Terminate.**



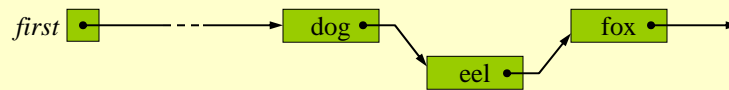
20

## SLL insertion (3)

- **Animation (insertion after intermediate node):**

To insert *elem* at a given point in the SLL headed by *first*:

1. Make *ins* a link to a newly-created node with element *elem* and successor null.
2. If the insertion point is before the first node:
  - 2.1. Set node *ins*'s successor to *first*.
  - 2.2. Set *first* to *ins*.
3. If the insertion point is after the node *pred*:
  - 3.1. Set node *ins*'s successor to node *pred*'s successor.
  - 3.2. Set node *pred*'s successor to *ins*.
4. **Terminate.**



21

## SLL insertion (4)

- **Implementation as a Java method (in class SLL):**

```

public void insert (Object elem SLLNode pred) {
// Insert elem at a given point in this SLL, either after the
// node
// pred, or before the first node if pred is null.
SLLNode ins = new SLLNode(elem, null);
if (pred == null) {
ins.succ = this.first;
this.first = ins;
} else {
ins.succ = pred.succ;
pred.succ = ins;
}
}
  
```

22

## DLL insertion (1)

- **DLL insertion algorithm:**

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. Terminate.

23

## DLL insertion (2)

- **Auxiliary forward SLL insertion algorithm:**

To insert node *ins* at a given point in the forward SLL headed by *first*:

1. If the insertion point is before the first node:
  - 1.1. Set node *ins*'s successor to *first*.
  - 1.2. Set *first* to *ins*.
2. If the insertion point is after the node *pred*:
  - 2.1. Set node *ins*'s successor to node *pred*'s successor.
  - 2.2. Set node *pred*'s successor to *ins*.
3. Terminate.

24

## DLL insertion (3)

- **Auxiliary backward SLL insertion algorithm:**

To insert node *ins* after node *succ* in the backward SLL headed by *last*:

1. If *succ* is null:
  - 1.1. Set node *ins*'s predecessor to *first*.
  - 1.2. Set *last* to *ins*.
2. If *succ* is not null:
  - 2.1. Set node *ins*'s predecessor to node *succ*'s predecessor.
  - 2.2. Set node *succ*'s predecessor to *ins*.
3. Terminate.

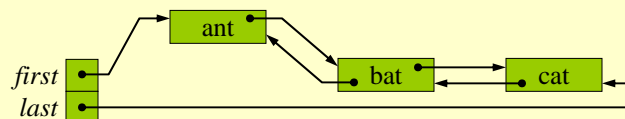
25

## DLL insertion (4)

- **Animation (insertion before the first node):**

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. **Terminate.**



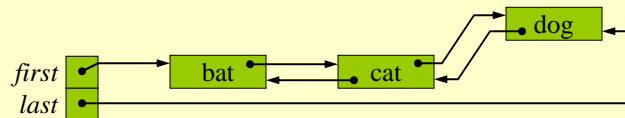
26

## DLL insertion (5)

- **Animation (insertion after the last node):**

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. **Terminate.**



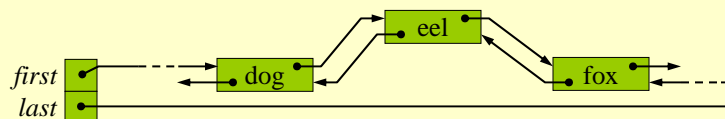
27

## DLL insertion (6)

- **Animation (insertion between nodes):**

To insert *elem* at a given point in the DLL headed by (*first*, *last*):

1. Make *ins* a link to a newly-created node with element *elem*, predecessor null, and successor null.
2. Insert *ins* at the insertion point in the forward SLL headed by *first*.
3. Let *succ* be *ins*'s successor (or null if *ins* has no successor).
4. Insert *ins* after node *succ* in the backward SLL headed by *last*.
5. **Terminate.**



28

## Deletion

- **Problem: Delete a given node from a linked list.**
- **Four cases to consider:**
  - 1) deletion of a singleton node;
  - 2) deletion of the first (but not last) node;
  - 3) deletion of the last (but not first) node;
  - 4) deletion of an intermediate node.
- **The deletion algorithm needs links to the deleted node's successor and predecessor.**

29

## SLL deletion (1)

- **SLL deletion algorithm:**

To delete node *del* from the SLL headed by *first*:

  1. Let *succ* be node *del*'s successor.
  2. If *del = first*:
    - 2.1. Set *first* to *succ*.
  3. Otherwise (if *del ≠ first*):
    - 3.1. Let *pred* be node *del*'s predecessor.
    - 3.2. Set node *pred*'s successor to *succ*.
  4. Terminate.
- **But there is no link from node *del* to its predecessor, so step 3.1 can access *del*'s predecessor only by following links from *first*!**

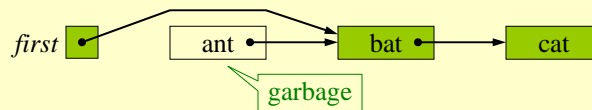
30

## SLL deletion (2)

- Animation (deleting the first node):

To delete node *del* from the SLL headed by *first*:

1. Let *succ* be node *del*'s successor.
2. If *del* = *first*:
  - 2.1. Set *first* to *succ*.
3. Otherwise (if *del* ≠ *first*):
  - 3.1. Let *pred* be node *del*'s predecessor.
  - 3.2. Set node *pred*'s successor to *succ*.
4. **Terminate.**



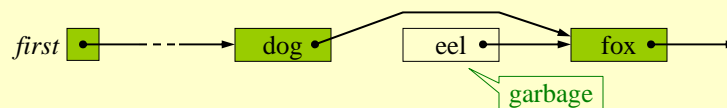
31

## SLL deletion (3)

- Animation (deleting an intermediate (or last) node):

To delete node *del* from the SLL headed by *first*:

1. Let *succ* be node *del*'s successor.
2. If *del* = *first*:
  - 2.1. Set *first* to *succ*.
3. Otherwise (if *del* ≠ *first*):
  - 3.1. Let *pred* be node *del*'s predecessor.
  - 3.2. Set node *pred*'s successor to *succ*.
4. **Terminate.**



32

## SLL deletion (4)

- **Analysis:**

Let  $n$  be the SLL's length.

**Step 3.1 must visit all nodes from the first node to the deleted node's predecessor. There are between 0 and  $n-1$  such nodes.**

**Average no. of nodes visited =  $(n-1)/2$**

**Time complexity is  $O(n)$ .**

33

## SLL deletion (5)

- **Implementation as a Java method (in class SLL):**

```
public void delete (SLLNode del) {  
    // Delete node del from this SLL.  
    SLLNode succ = del.succ;  
    if (del == this.first) {  
        this.first = succ;  
    } else {  
        SLLNode pred = this.first;  
        while (pred.succ != del)  
            pred = pred.succ;  
        pred.succ = succ;  
    }  
}
```

34

## DLL deletion (1)

- **DLL deletion algorithm:**

To delete node *del* from the DLL headed by (*first*, *last*):

1. Let *pred* and *succ* be node *del*'s predecessor and successor.
2. Delete node *del*, whose predecessor is *pred*, from the forward SLL headed by *first*.
3. Delete node *del*, whose successor is *succ*, from the backward SLL headed by *last*.
4. Terminate.

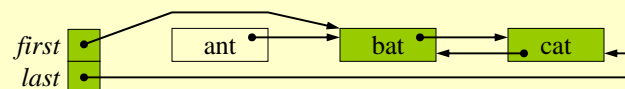
35

## DLL deletion (2)

- **Animation (deleting the first (but not last) node):**

To delete node *del* from the DLL headed by (*first*, *last*):

1. Let *pred* and *succ* be node *del*'s predecessor and successor.
2. Delete node *del*, whose predecessor is *pred*, from the forward SLL headed by *first*.
3. Delete node *del*, whose successor is *succ*, from the backward SLL headed by *last*.
4. **Terminate.**



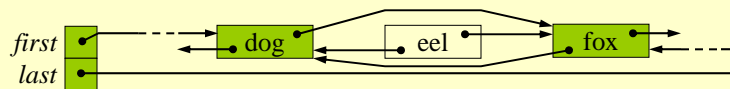
36

## DLL deletion (3)

- **Animation (deleting an intermediate node):**

To delete node *del* from the DLL headed by (*first*, *last*):

1. Let *pred* and *succ* be node *del*'s predecessor and successor.
2. Delete node *del*, whose predecessor is *pred*, from the forward SLL headed by *first*.
3. Delete node *del*, whose successor is *succ*, from the backward SLL headed by *last*.
4. **Terminate.**



37

## Comparison of insertion and deletion algorithms

Algorithm	SLL	DLL
Insertion	$O(1)$	$O(1)$
Deletion	$O(n)$	$O(1)$

38

## Searching (1)

- **Problem:** Search for a given target value in a linked list.
- **Unsorted SLL linear search algorithm:**  
To find which (if any) node of the SLL headed by *first* contains an element equal to *target*:
  1. For each node *curr* in the SLL headed by *first*, repeat:
    - a. If *target* is equal to node *curr*'s element, terminate with answer *curr*.
  2. Terminate with answer *none*.
- **DLL linear search is similar, except that we can search from last to first if preferred.**

39

## Searching (2)

- **Analysis (counting comparisons):**  
Let  $n$  be the SLL's length.
- If the search is **successful**:  
Average no. of comparisons =  $(n + 1)/2$
- If the search is **unsuccessful**:  
No. of comparisons =  $n$
- In either case, time complexity is  $O(n)$ .

40

## Searching (3)

- **Java implementation:**

```
public SLLNode search (Object target) {  
    // Find which (if any) node of this SLL contains an  
    // element equal to  
    // target. Return a link to the matching node (or null if  
    // there is  
    // none).  
    for (SLLNode curr = this.first;  
        curr != null; curr = curr.succ) {  
        if (target.equals(curr.element))  
            return curr;  
    }  
    return null;  
}
```

41