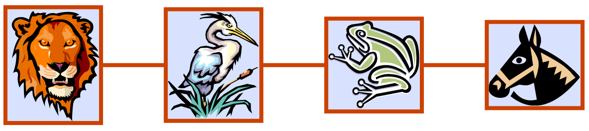
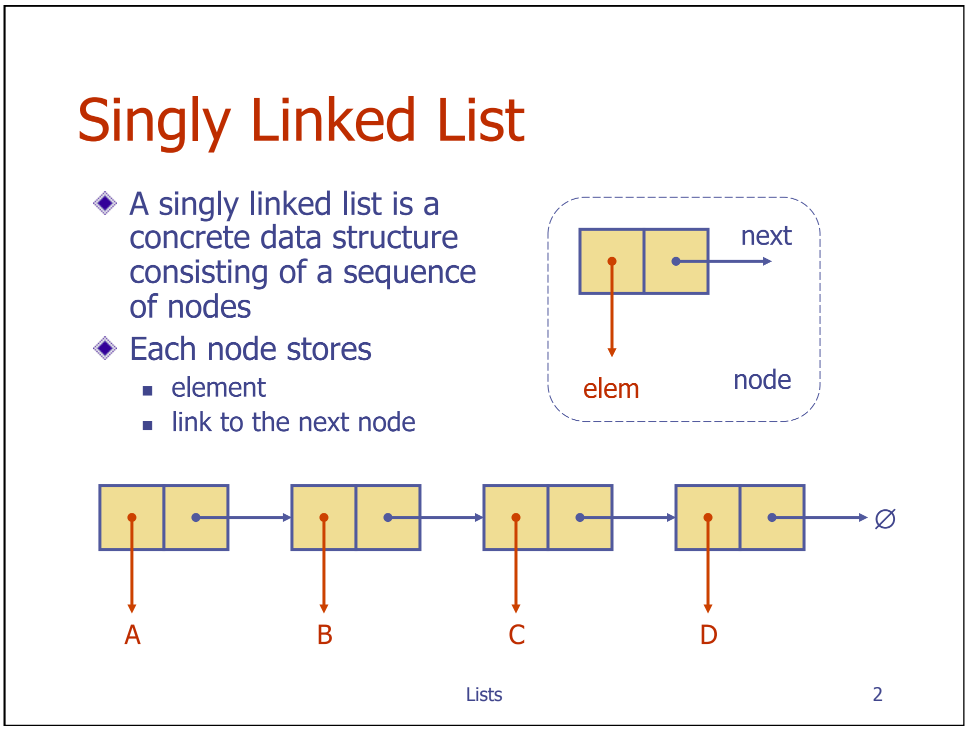
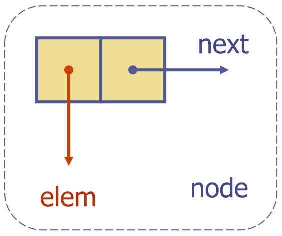
Linked Lists

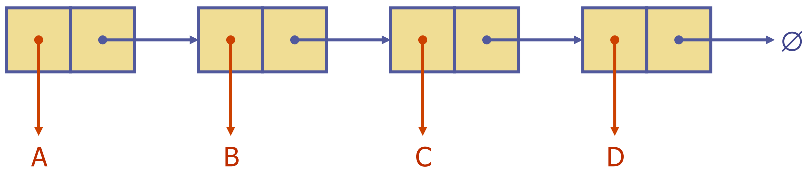


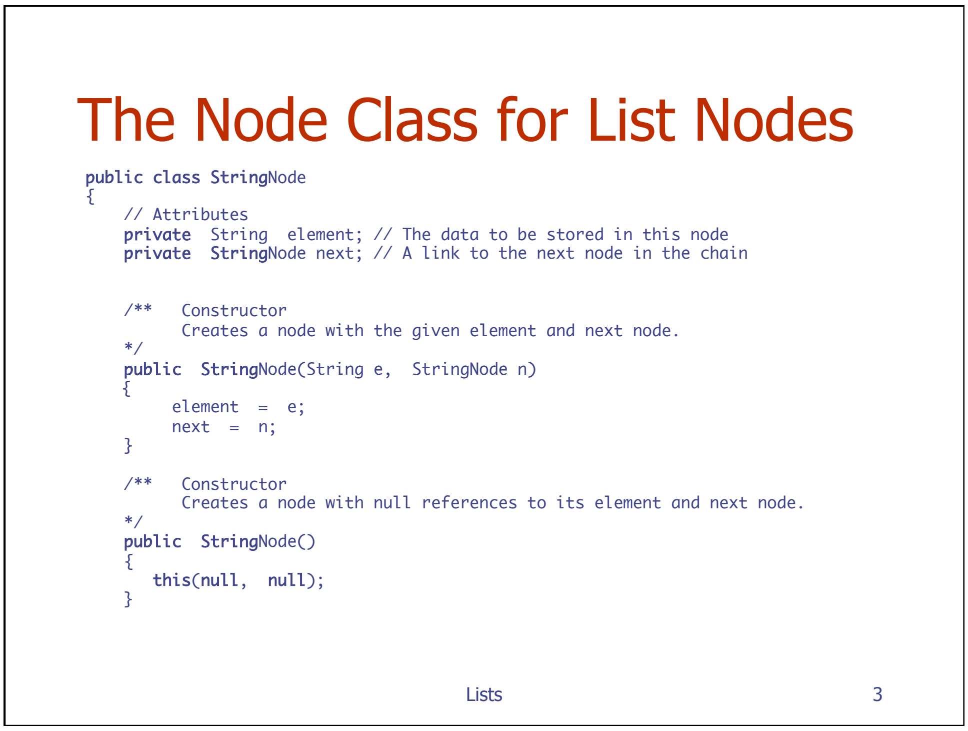


Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node



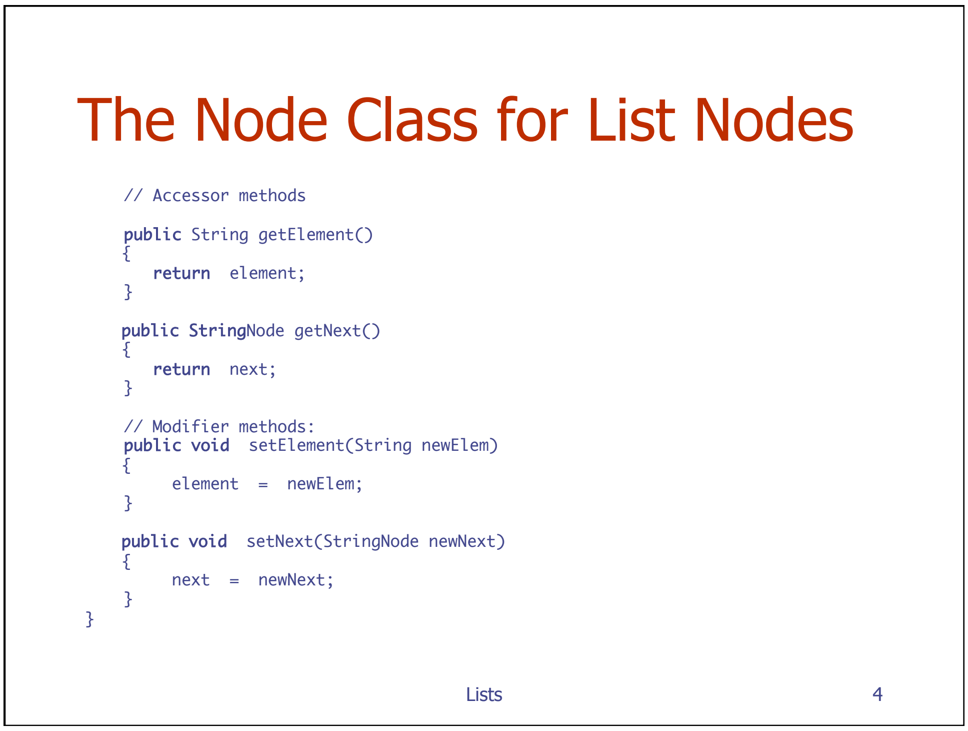




The Node Class for List Nodes

```
public class StringNode
{
    // Attributes
    private  String  element; // The data to be stored in this node
    private  StringNode next; // A link to the next node in the chain


    /**   Constructor
          Creates a node with the given element and next node.
    */
    public  StringNode(String e,  StringNode n)
    {
         element  =  e;
         next  =  n;
    }

    /**   Constructor
          Creates a node with null references to its element and next node.
    */
    public  StringNode()
    {
       this(null,  null);
    }
```

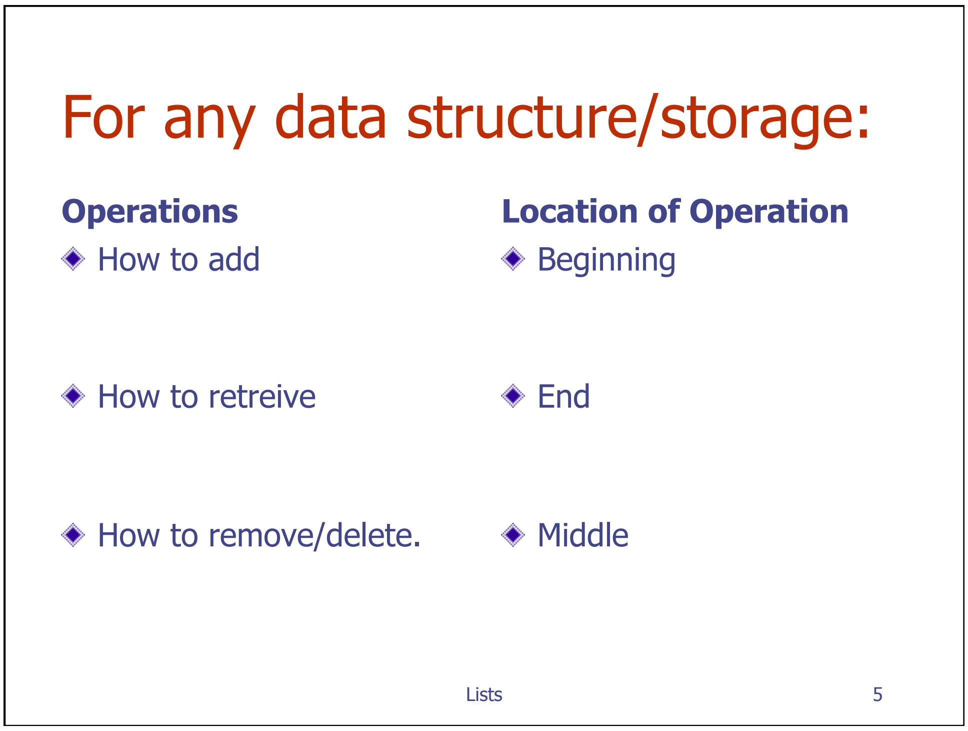
Lists 3



The Node Class for List Nodes

```
    // Accessor methods

    public String getElement()
    {
       return  element;
    }

    public StringNode getNext()
    {
       return  next;
    }

    // Modifier methods:
    public void  setElement(String newElem)
    {
         element  =  newElem;
    }

    public void  setNext(StringNode newNext)
    {
         next  =  newNext;
    }
}
```

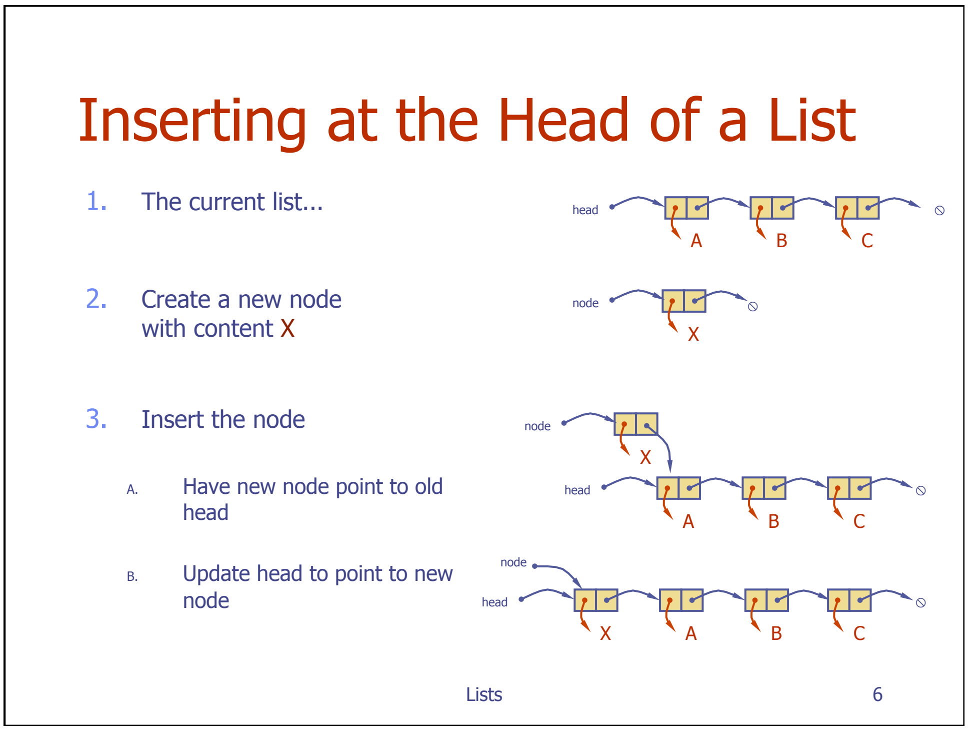
Lists 4



For any data structure/storage:

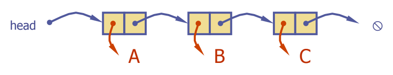
Operations
- How to add
- How to retreive
- How to remove/delete.

Location of Operation
- Beginning
- End
- Middle

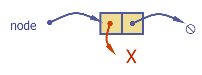


Inserting at the Head of a List

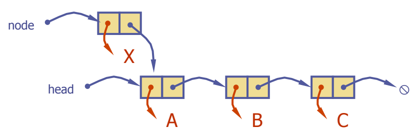
1. The current list...



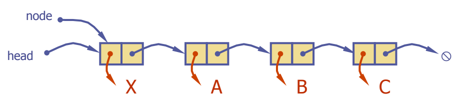
2. Create a new node with content X

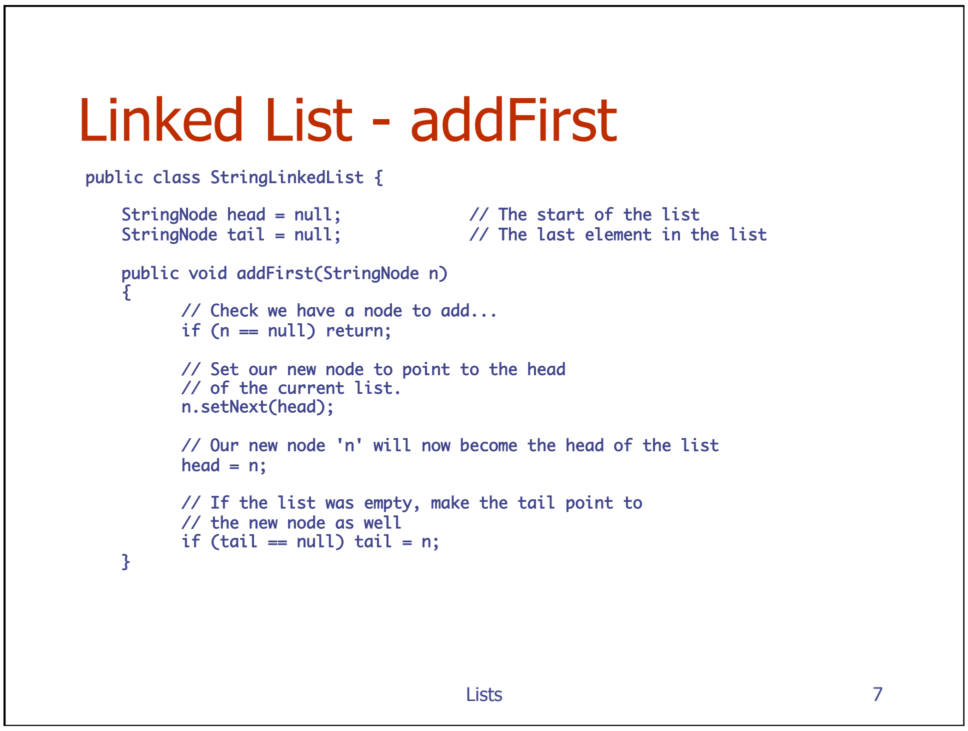


3. Insert the node
   - A. Have new node point to old head



   - B. Update head to point to new node

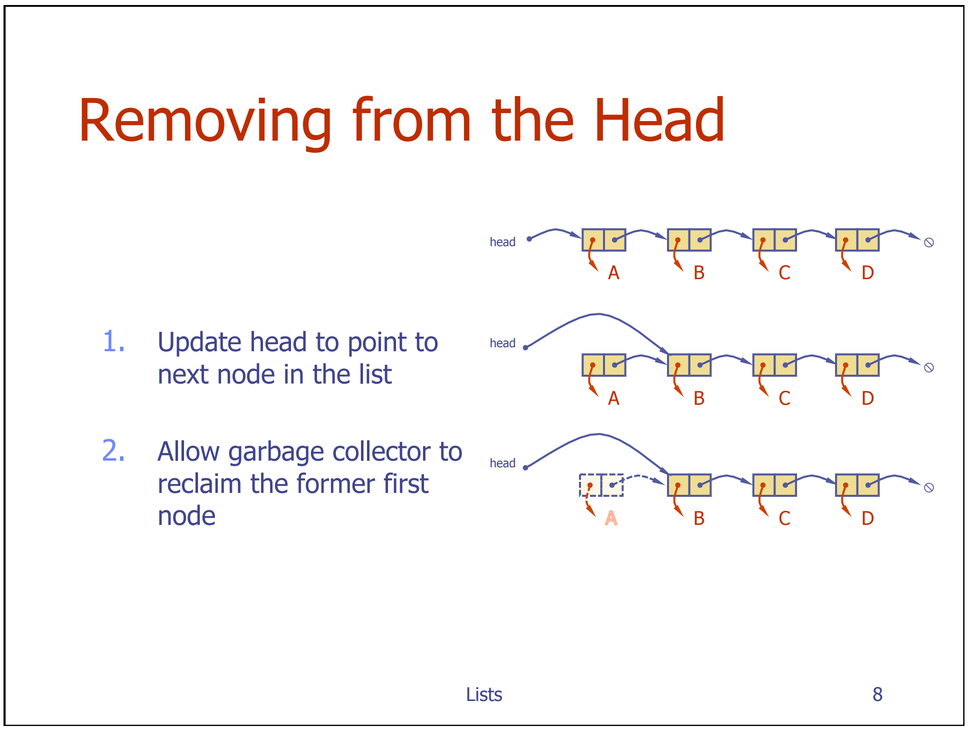




Linked List - addFirst

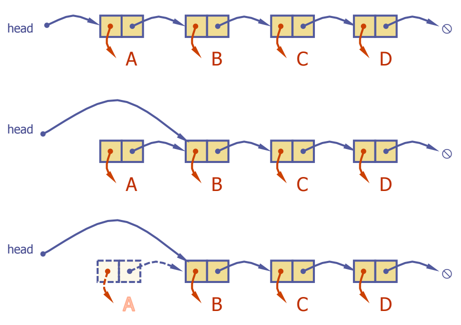
```
public class StringLinkedList {

    StringNode head = null;              // The start of the list
    StringNode tail = null;              // The last element in the list

    public void addFirst(StringNode n)
    {
          // Check we have a node to add...
          if (n == null) return;

          // Set our new node to point to the head
          // of the current list.
          n.setNext(head);

          // Our new node 'n' will now become the head of the list
          head = n;

          // If the list was empty, make the tail point to
          // the new node as well
          if (tail == null) tail = n;
    }
```

Lists 7

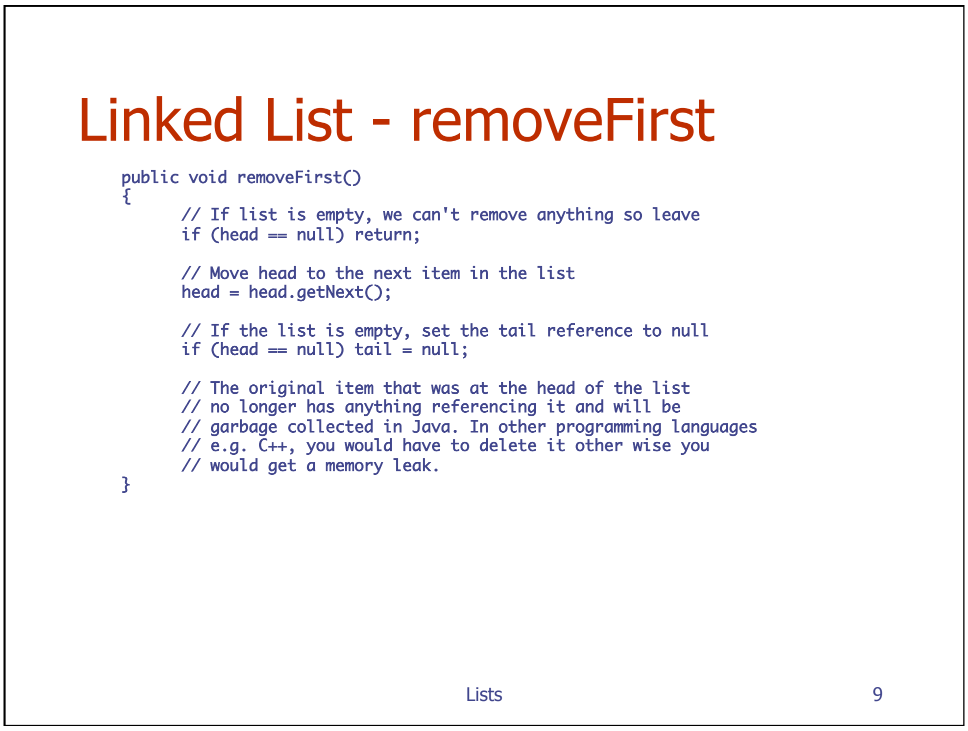


Removing from the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



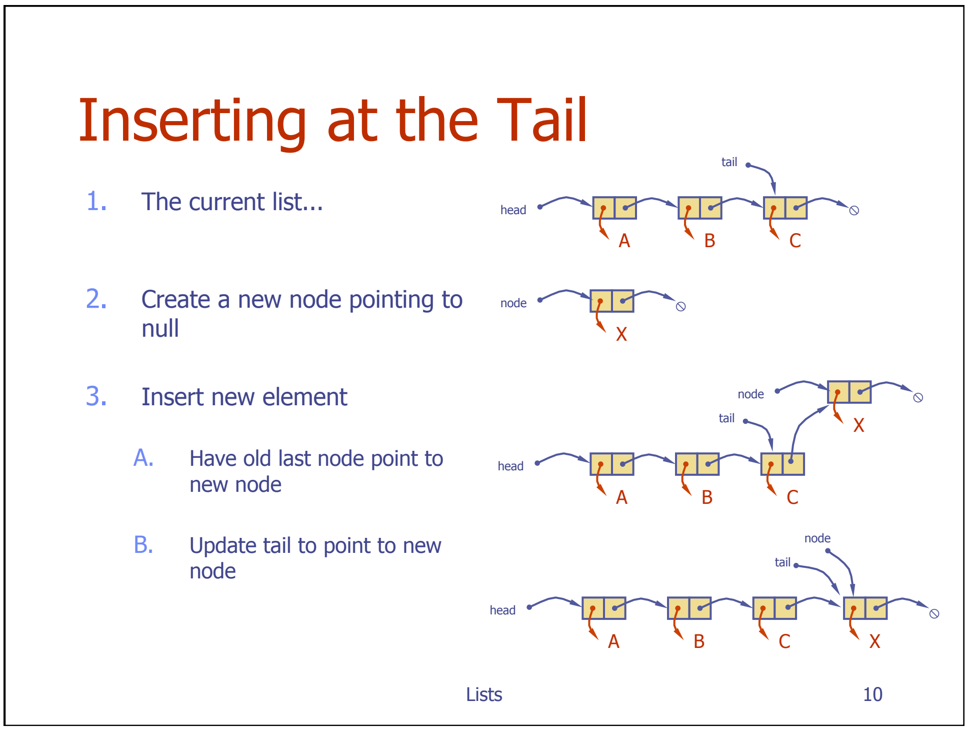
Lists 8



Linked List - removeFirst

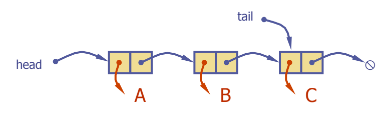
```
public void removeFirst()
{
        // If list is empty, we can't remove anything so leave
        if (head == null) return;

        // Move head to the next item in the list
        head = head.getNext();

        // If the list is empty, set the tail reference to null
        if (head == null) tail = null;

        // The original item that was at the head of the list
        // no longer has anything referencing it and will be
        // garbage collected in Java. In other programming languages
        // e.g. C++, you would have to delete it other wise you
        // would get a memory leak.
}
```

Lists 9

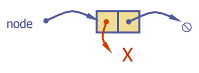


Inserting at the Tail

1. The current list...

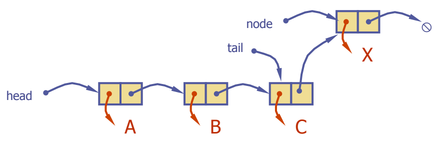


2. Create a new node pointing to null

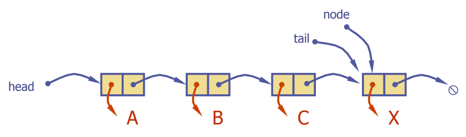


3. Insert new element

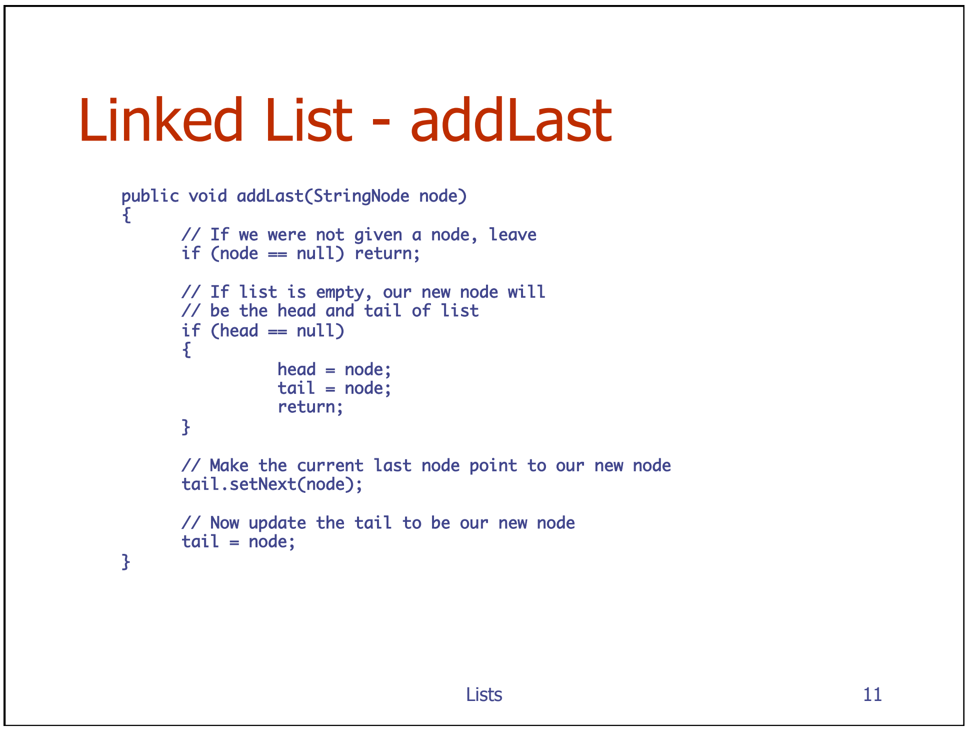
   A. Have old last node point to new node



   B. Update tail to point to new node



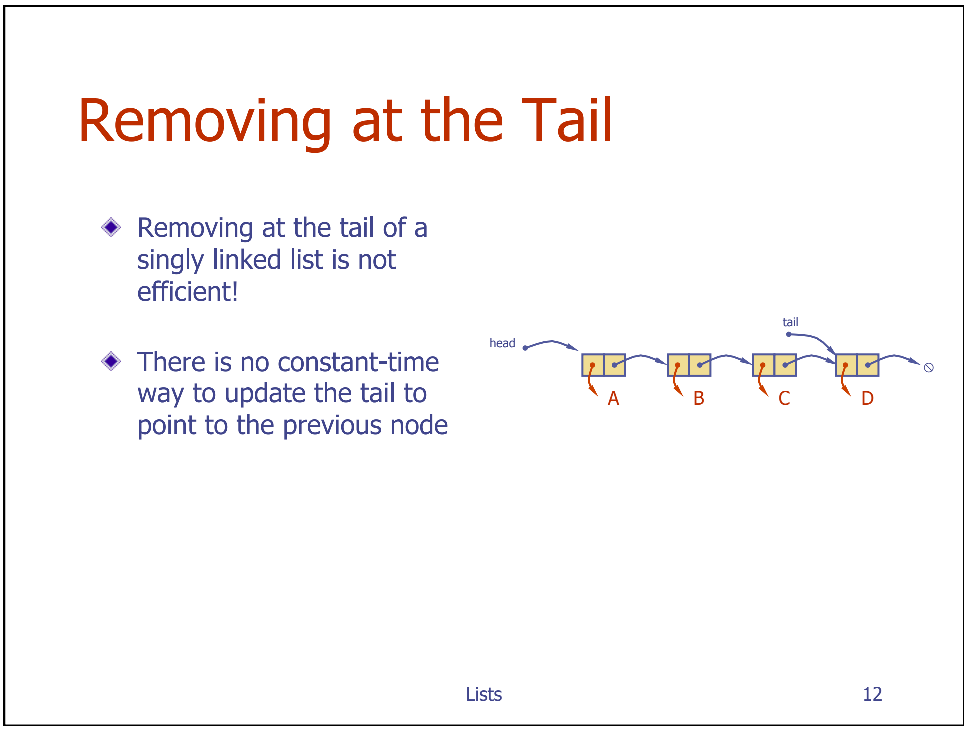
Lists 10



Linked List - addLast

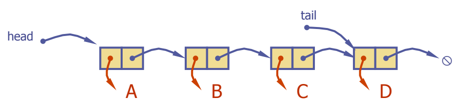
```
public void addLast(StringNode node)
{
      // If we were not given a node, leave
      if (node == null) return;

      // If list is empty, our new node will
      // be the head and tail of list
      if (head == null)
      {
                head = node;
                tail = node;
                return;
      }

      // Make the current last node point to our new node
      tail.setNext(node);

      // Now update the tail to be our new node
      tail = node;
}
```

Lists 11



Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



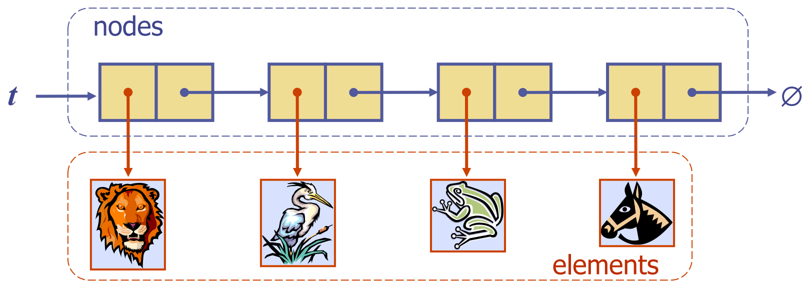
Lists 12

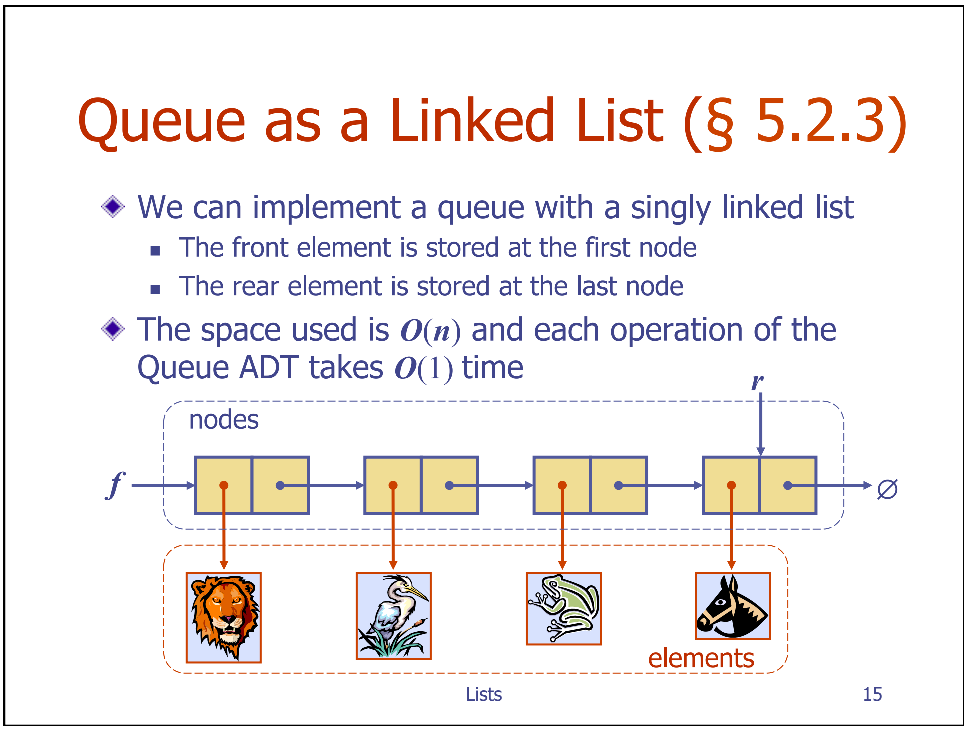
Linked List - removeLast

```
public void removeLast() {
      if (head == null) return; // If list is empty, leave

      // If head is also the tail, the list
      // will be empty
      if (head == tail) {
                head = null;
                tail = null;
                return;
      }

      // Start at the head of the list
      StringNode n = head;

      // Now look for the last item
      while (n.getNext() != tail)
                n = n.getNext();

      // n should now be pointing to the last but one
      // node in the list. This will be the new tail
      // We are going to drop the last element in the list
      // so make the current node's next pointer null
      n.setNext(null);

      // The old tail node is now replaced with 'n'. The
      // old tail node has no reference and will be garbage
      tail = n;
}
```

Stack as a Linked List (§ 5.1.3)

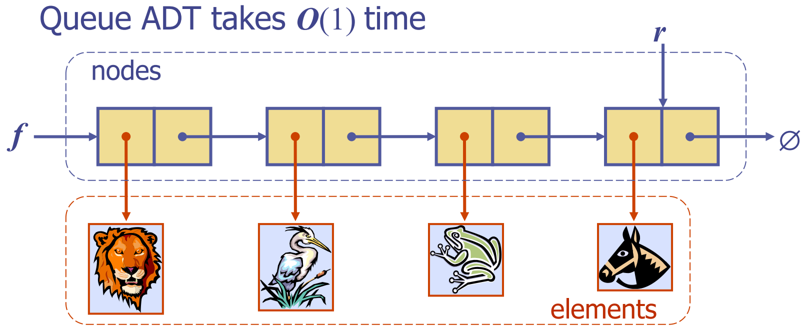
- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



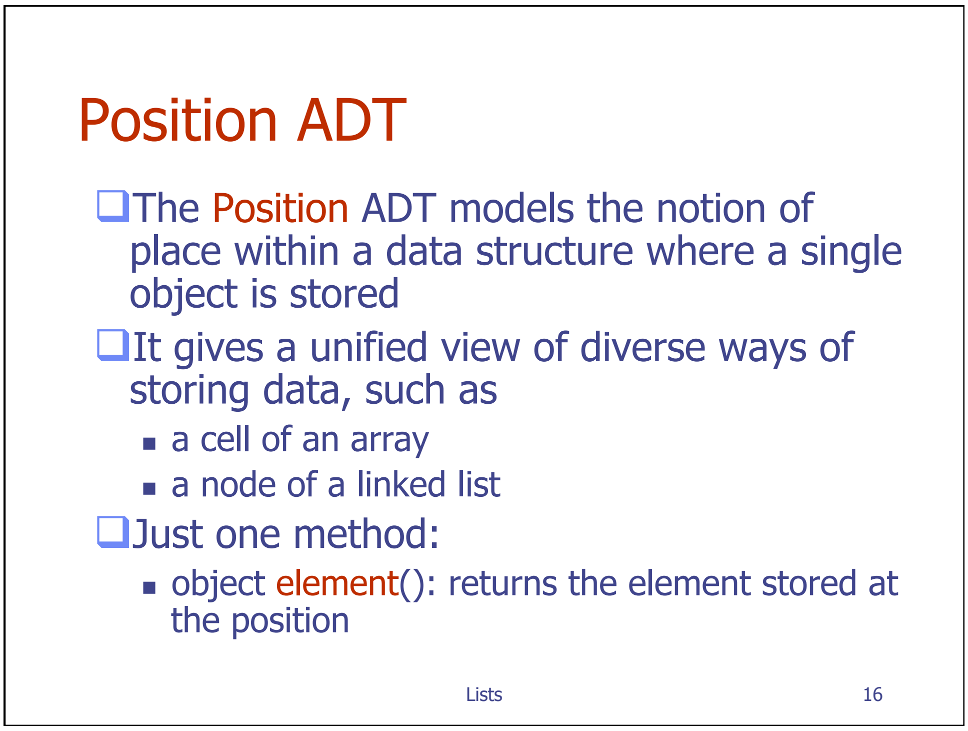


Queue as a Linked List (§ 5.2.3)

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Lists 15



Position ADT

- The Position ADT models the notion of place within a data structure where a single object is stored
- It gives a unified view of diverse ways of storing data, such as
  - a cell of an array
  - a node of a linked list
- Just one method:
  - object element(): returns the element stored at the position

Lists 16

Node List ADT

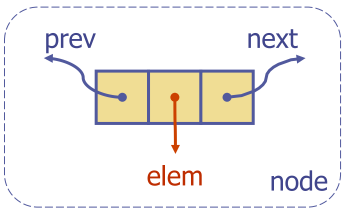
- The Node List ADT models a sequence of positions storing arbitrary objects
- It establishes a before/after relation between positions
- Generic methods:
  - size(), isEmpty()

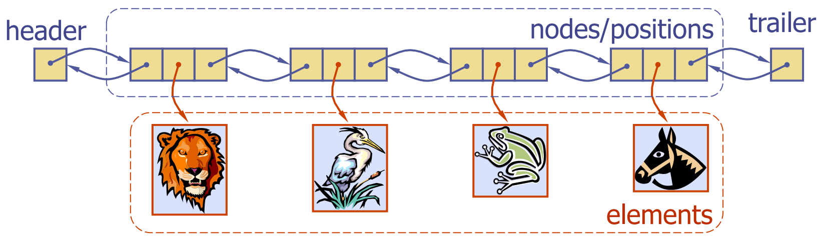
Accessor methods:
- first(), last()
- prev(p), next(p)

Update methods:
- set(p, e)
- addBefore(p, e), addAfter(p, e),
- addFirst(e), addLast(e)
- remove(p)

Doubly Linked List

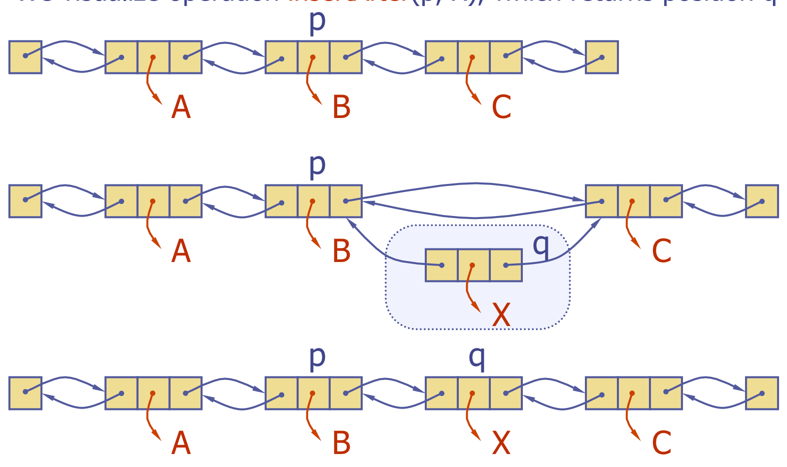
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Insertion

- We visualize operation insertAfter(p, X), which returns position q

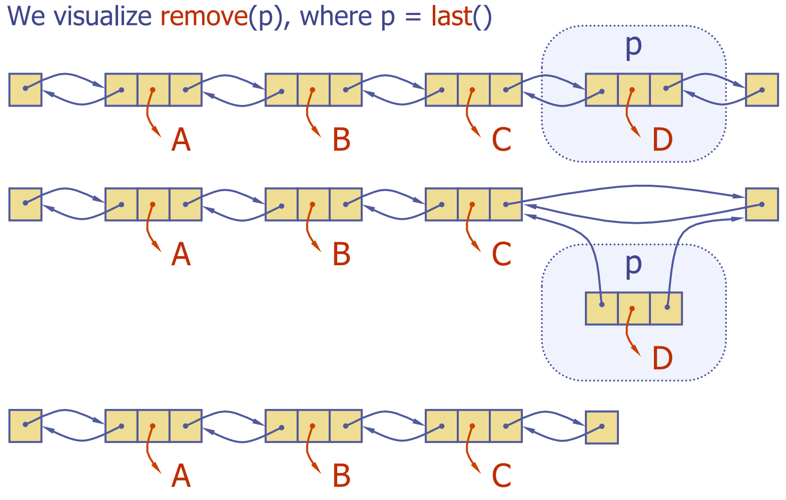


Insertion Algorithm

```
Algorithm addAfter(p,e):
    Create a new node v
    v.setElement(e)
    v.setPrev(p)            {link v to its predecessor}
    v.setNext(p.getNext())  {link v to its successor}
    (p.getNext()).setPrev(v) {link p’s old successor to v}
    p.setNext(v)            {link p to its new successor, v}
    return v   {the position for the element e}
```

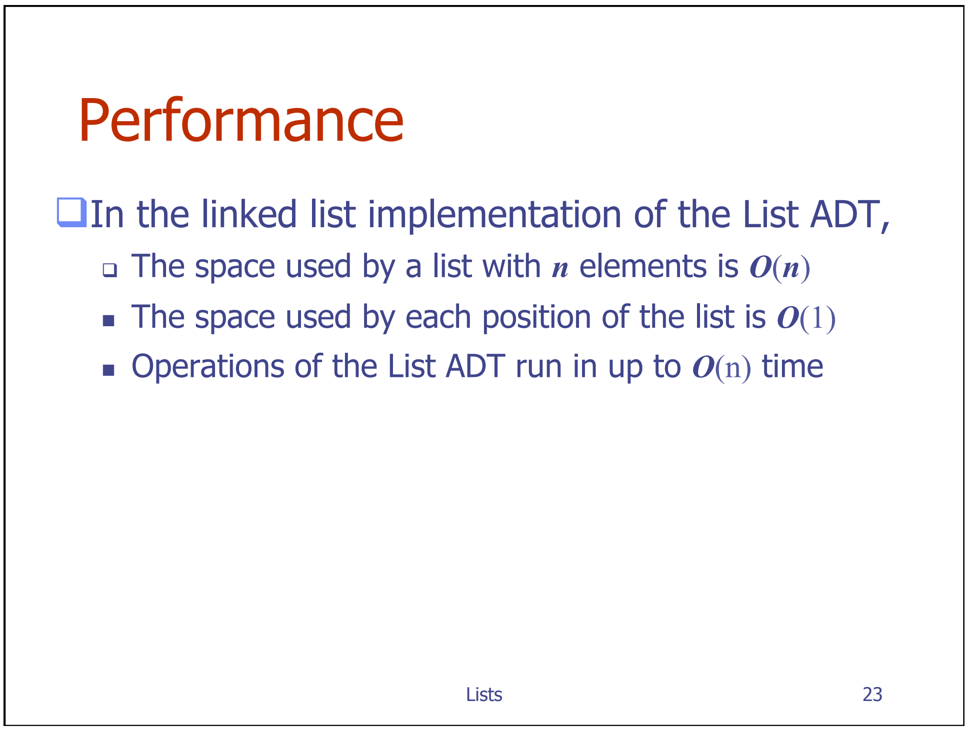
Deletion

- We visualize remove(p), where p = last()



Deletion Algorithm

```
Algorithm remove(p):
    t = p.element        {a temporary variable to hold the
                          return value}
    (p.getPrev()).setNext(p.getNext())     {linking out p}
    (p.getNext()).setPrev(p.getPrev())
    p.setPrev(null)      {invalidating the position p}
    p.setNext(null)
    return t
```



Performance

- In the linked list implementation of the List ADT,
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - Operations of the List ADT run in up to $O(n)$ time