Linked Structures

Lecture 12

Menu

- Testing collection implementations
- Queues
- Motivation for linked lists
- Linked structures for implementing Collections

Where have we been?

Implementing Collections with arrays:

- ArrayList: O(n) to add/remove, except at end
- Stack: O(1)
- ArraySet: O(n) (add/find/remove) (⇐ cost of searching)

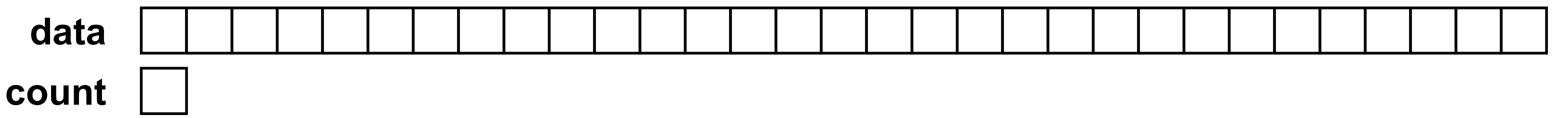
Testing Collection Implementations

- Write a **test method**
  - As part of the class or as a separate testing class
  - Should test all the operations
  - Should test normal and extreme cases
- Good practice:
  - write it first (**black box testing**)
  - implement the collection
  - extend the test method to cover the special cases of the implementation (**white box testing**)

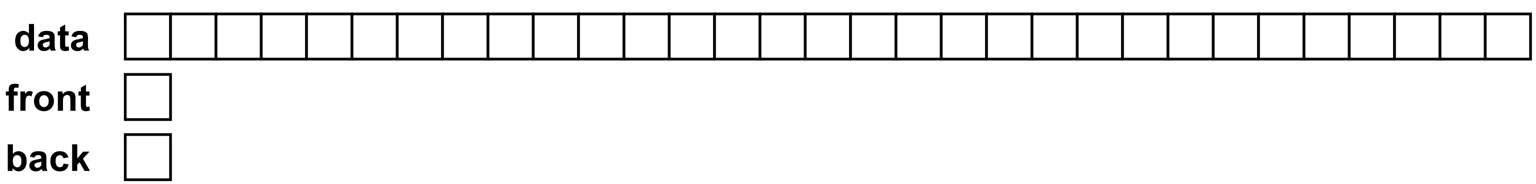
- Nicer design uses **tests/assertions**:
  - check that the code does the right thing
  - only report when there is a problem or error
- May take longer to write than the collection code!

Queues

- We haven't talked about implementing queues
- Simplest array implementations are slow:



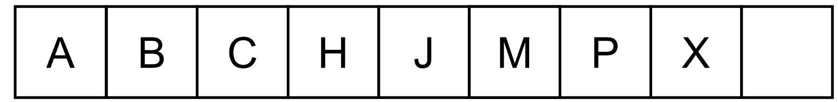
- Efficient array implementation
  - "wrapping around"
  - O(1) for add ("offer") and remove ("poll")



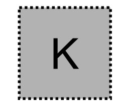
  - Have to be careful in ensureCapacity()
    - How do we know array is full?

How can we insert fast?

- Fast *access* in array



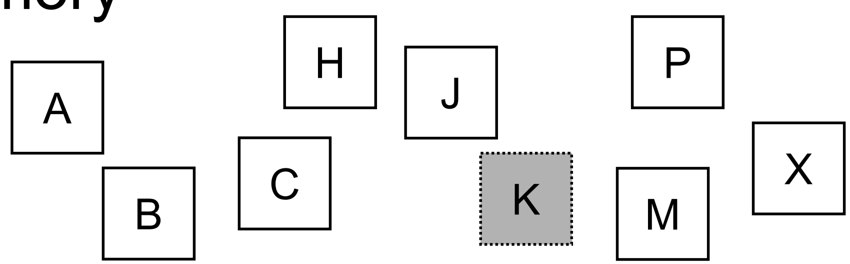
  ⇒ items must be sorted, to use binary search



- Arrays stored in contiguous chunks of memory.

  ⇒ inserting new items will be slow

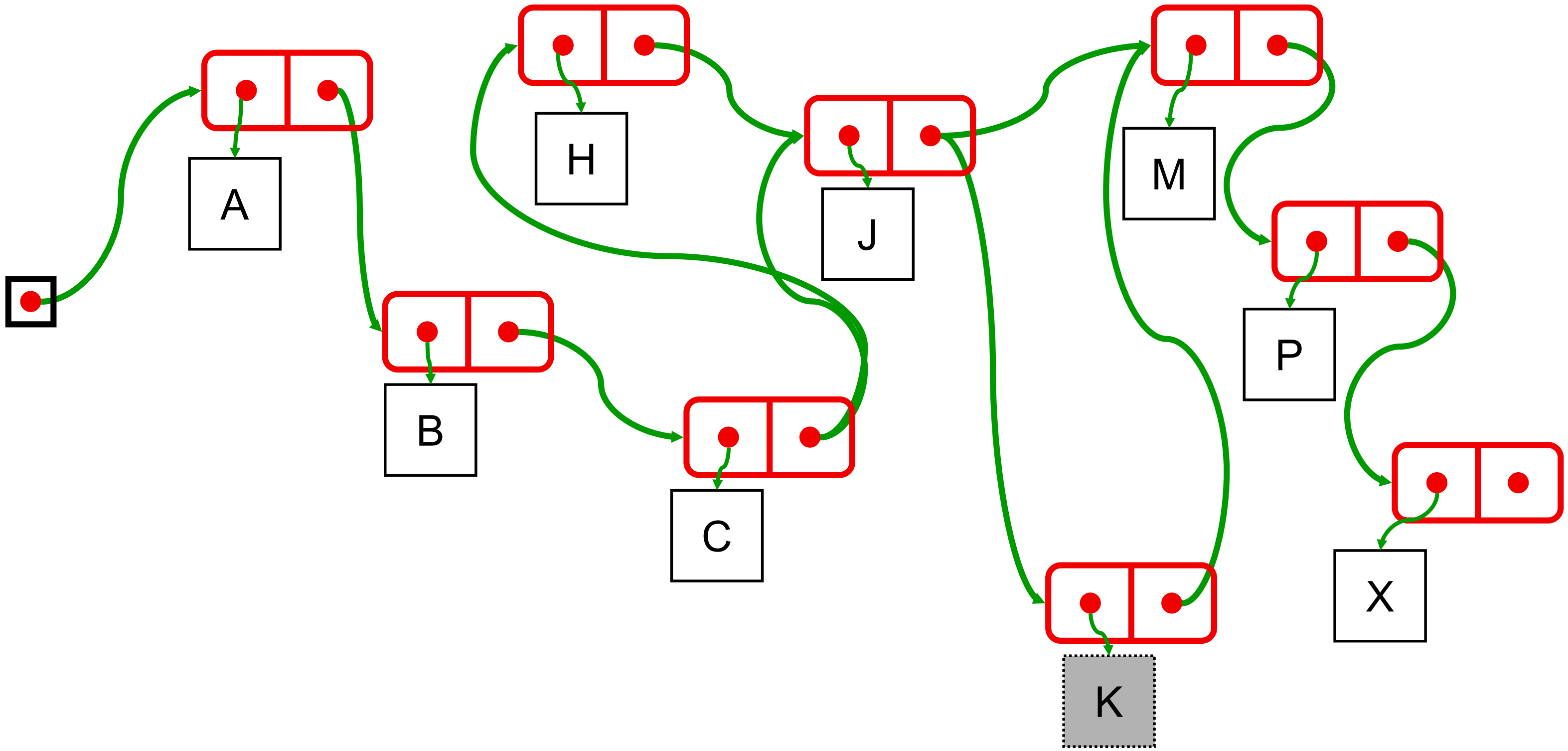
- Can’t insert fast with an array!
- To insert fast, we need each item to be in its own chunk of memory



- But, how do we keep track of the order?

Linked Structures

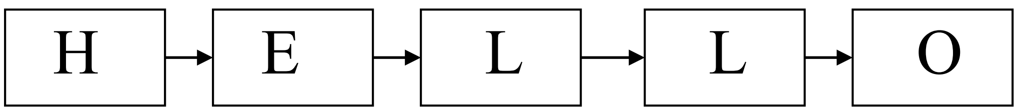
- Put each value in an object with a field for a link to the next



- Traverse the list by following the links
- Insert by changing links
- Remove by changing links

Linked lists

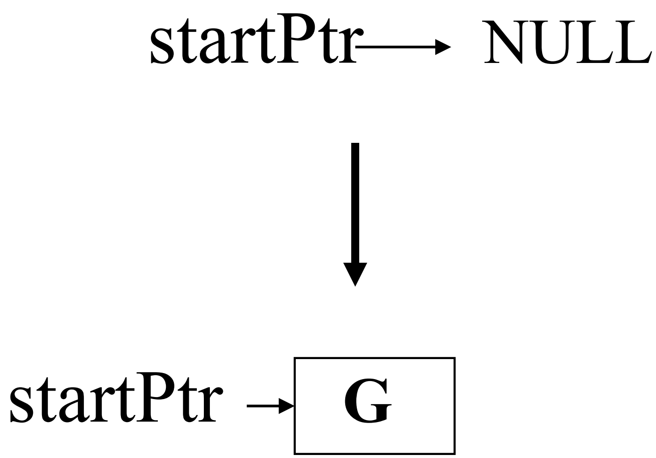
- collections of data in a row



- insertions & deletions anywhere in the list
- other operations: search for an element in the list, print the list, test if list empty, etc.

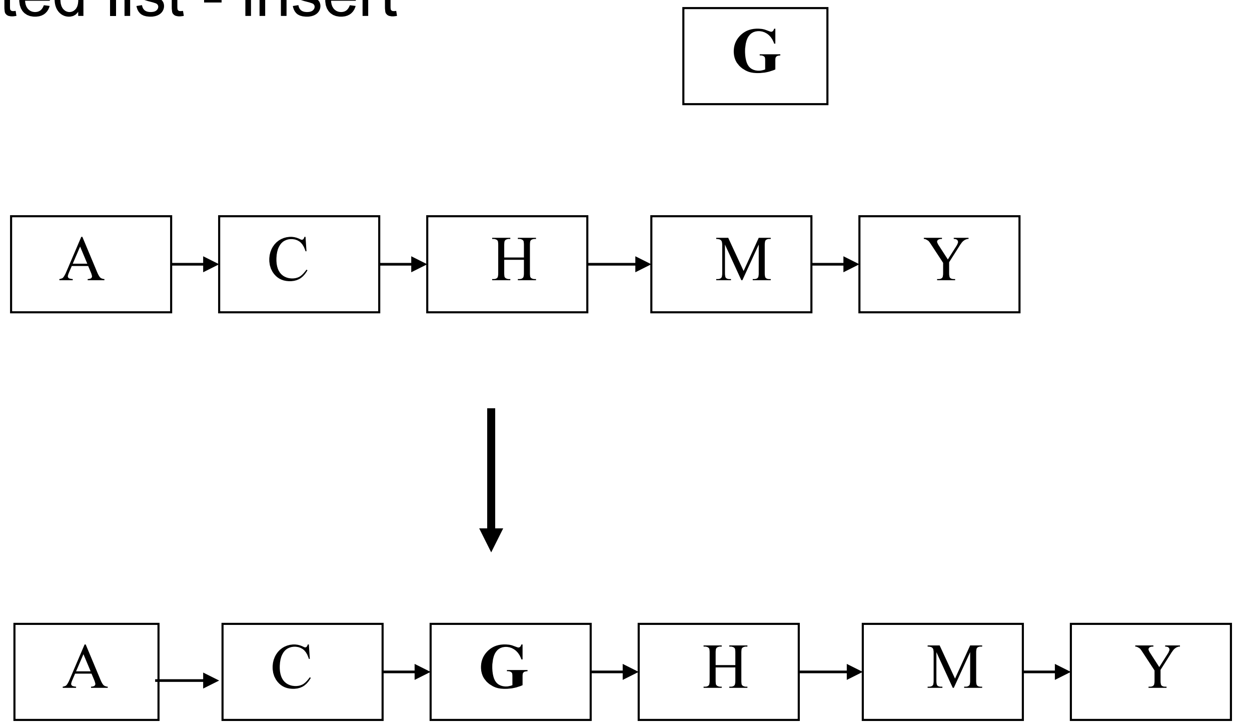
linked lists - insertion

- sorted list (empty) - insert $\boxed{G}$



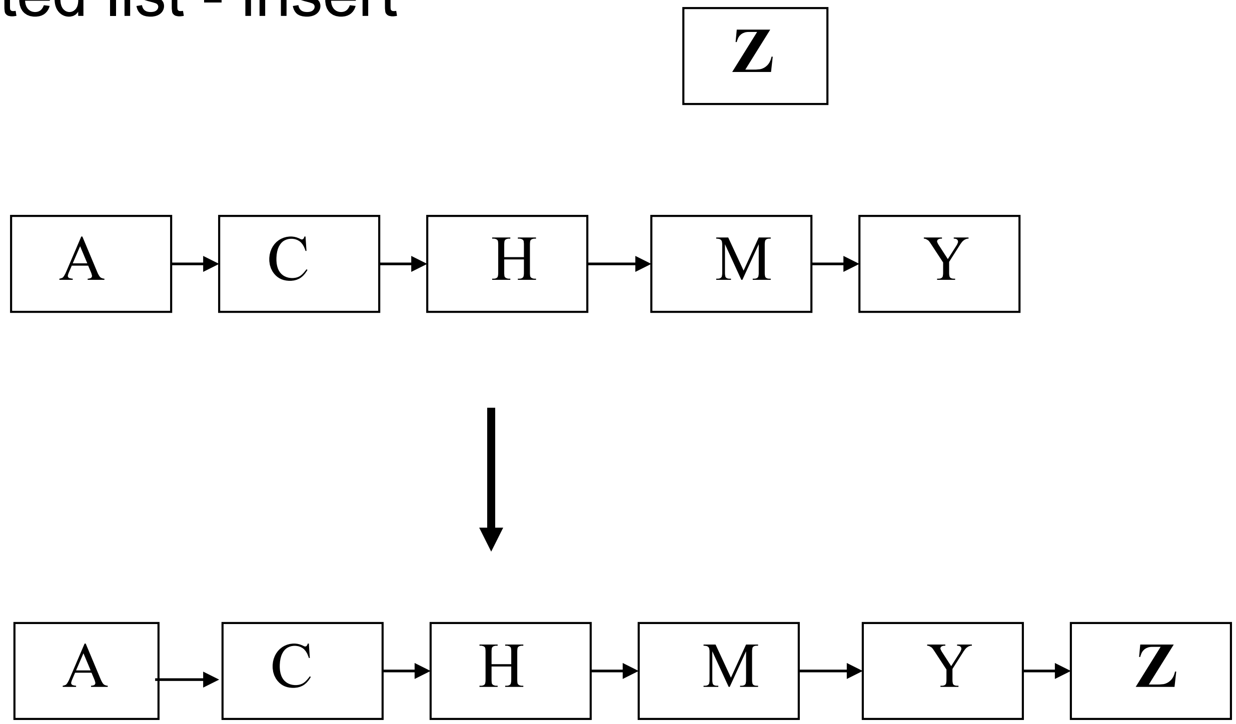
linked lists - insertion

- sorted list - insert

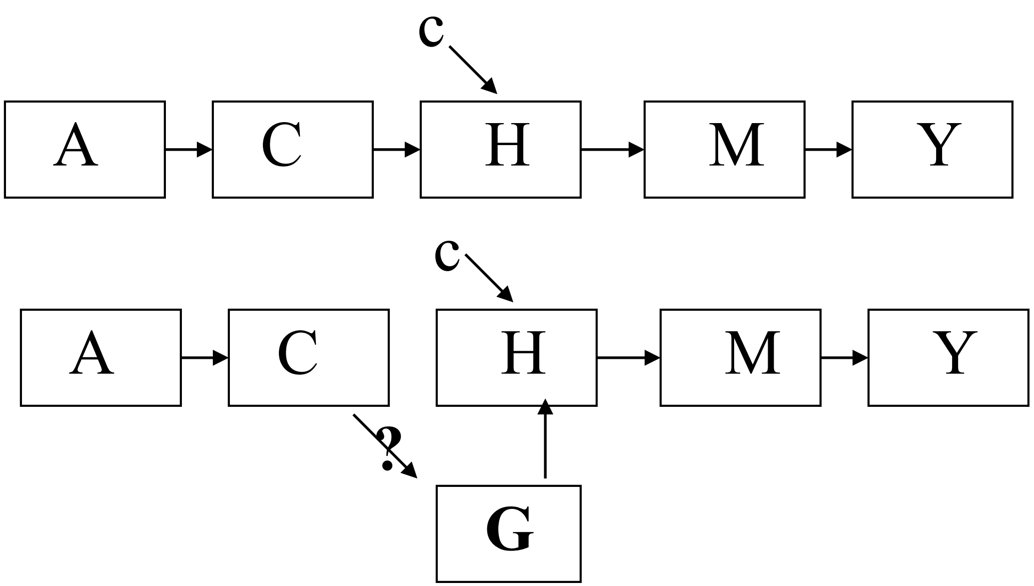


linked lists - insertion

- sorted list - insert

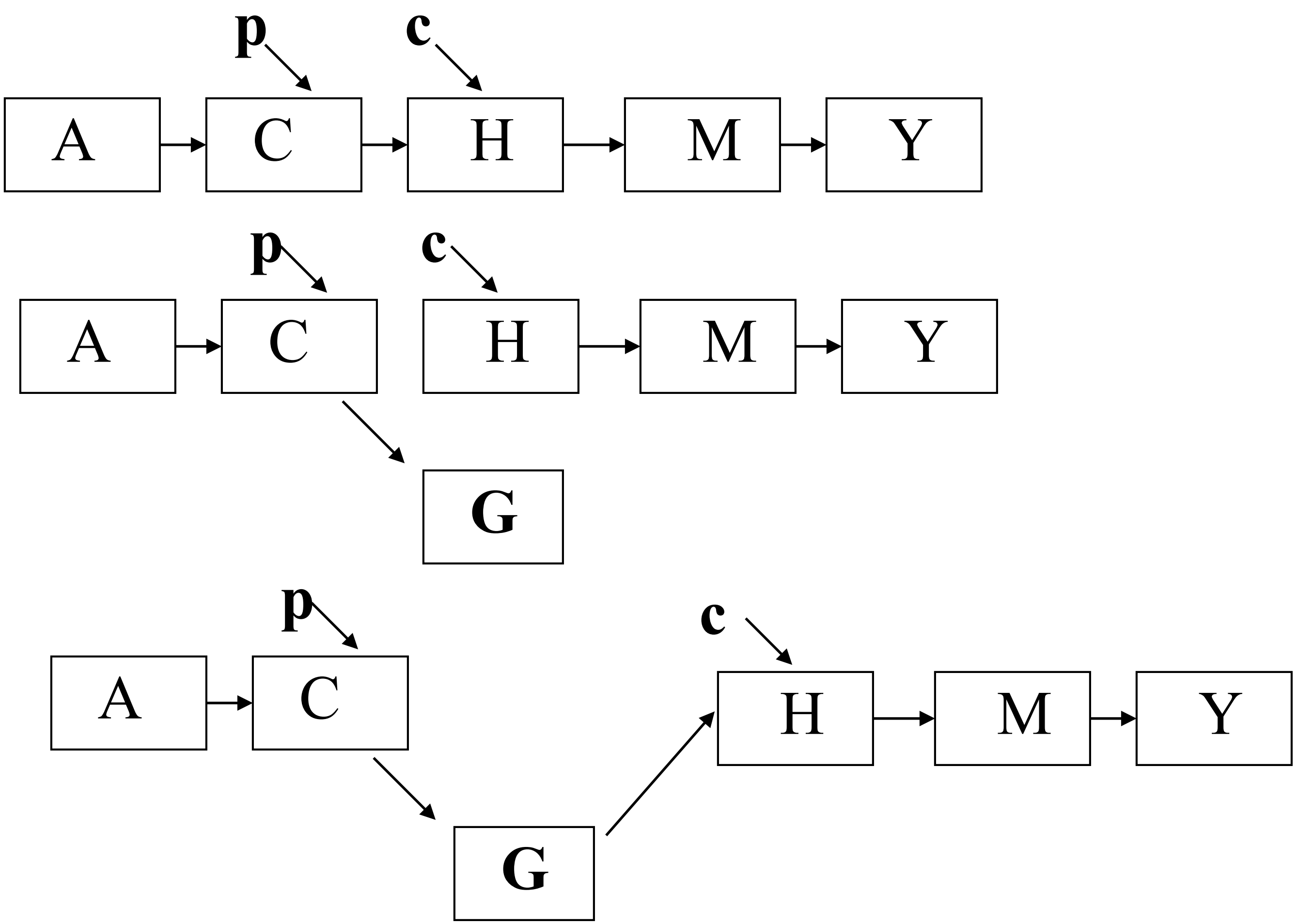


Insert in action



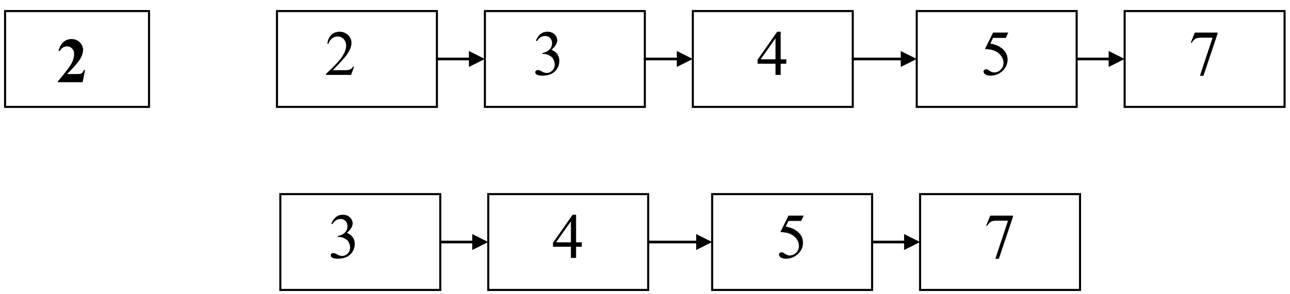
Problem: lost track of C when ‘H’ visited,
cannot redirect nextPtr in C to point to ‘G’
Solution ?

Insert in action

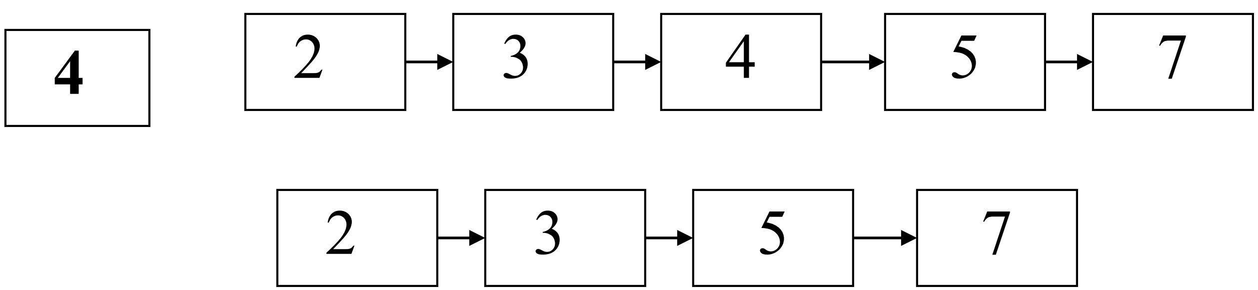


linked lists - search for deletion

- search & then delete



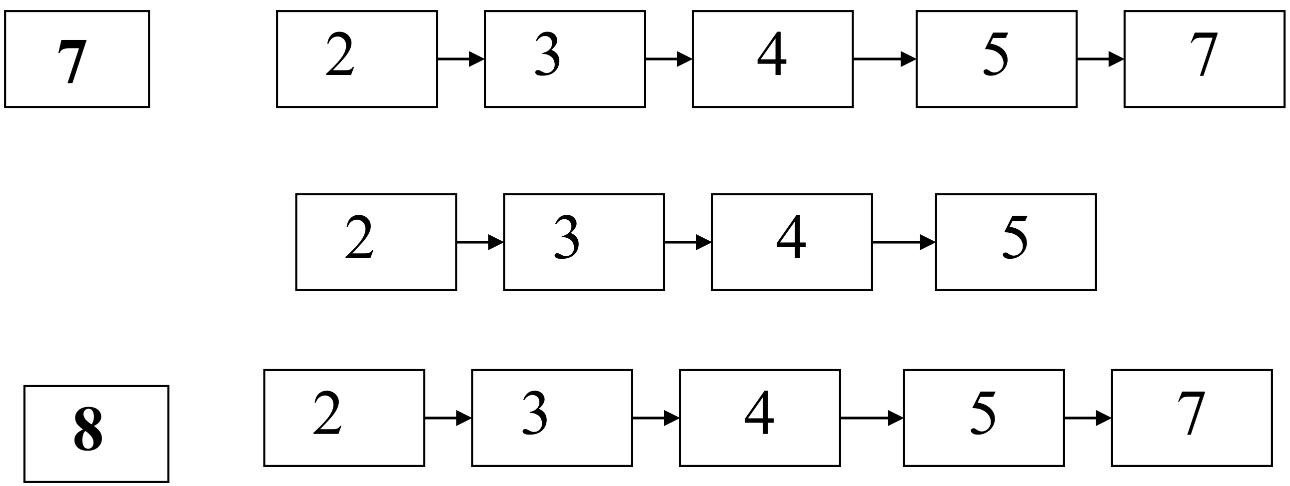
element in the front



element in the middle

linked lists - search for deletion

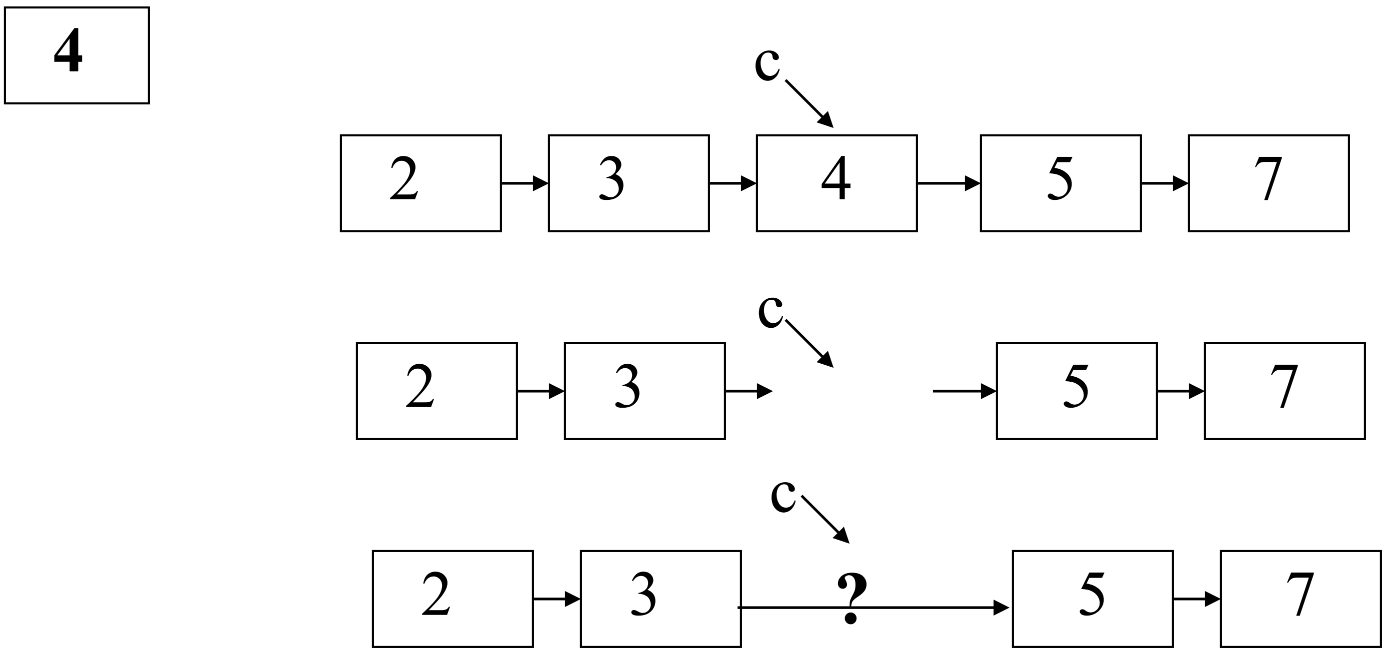
- search & then delete



element in the end

element not in the list

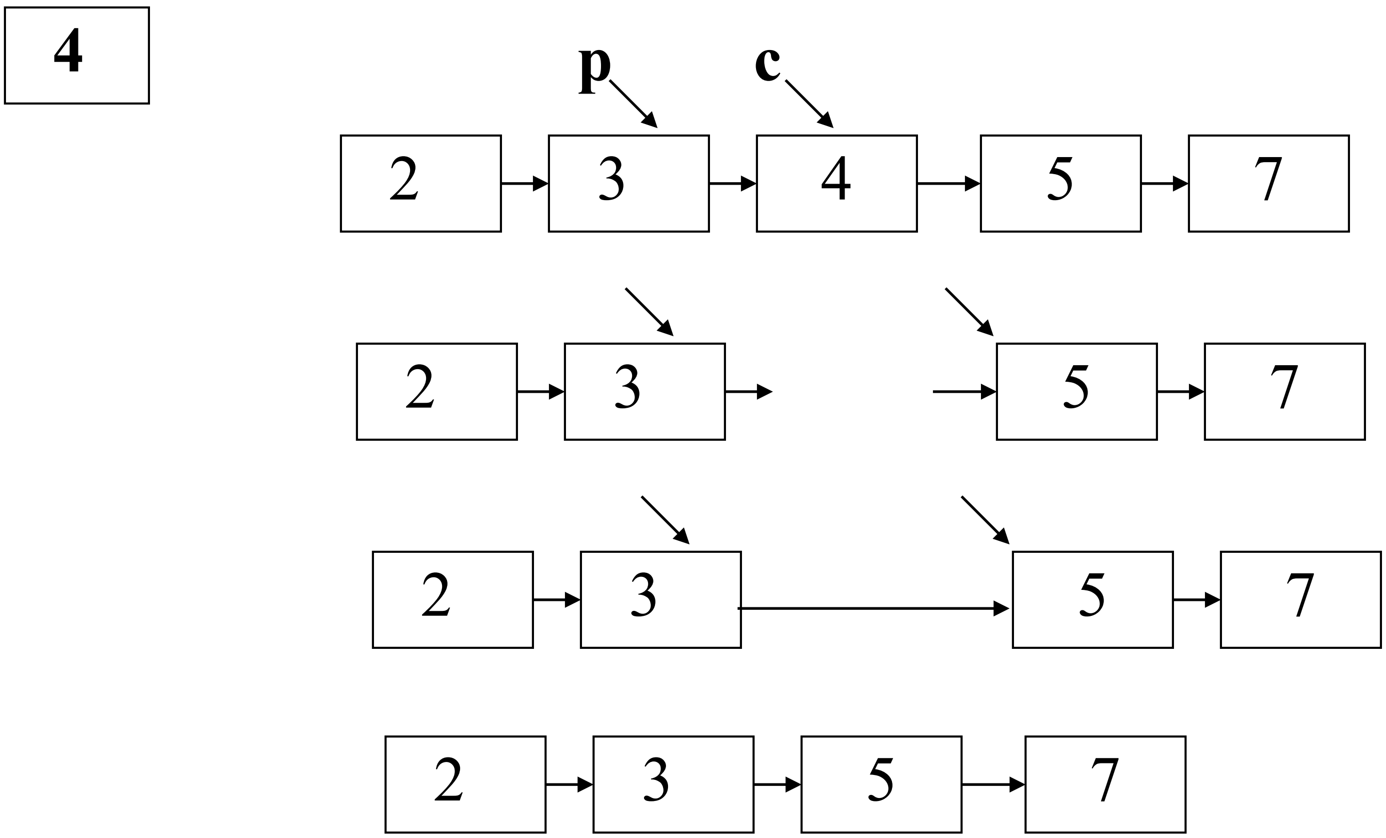
Delete in action



Problem: need to redirect nextPtr of 3 to 5, already lose track of 3 when c moved to 4

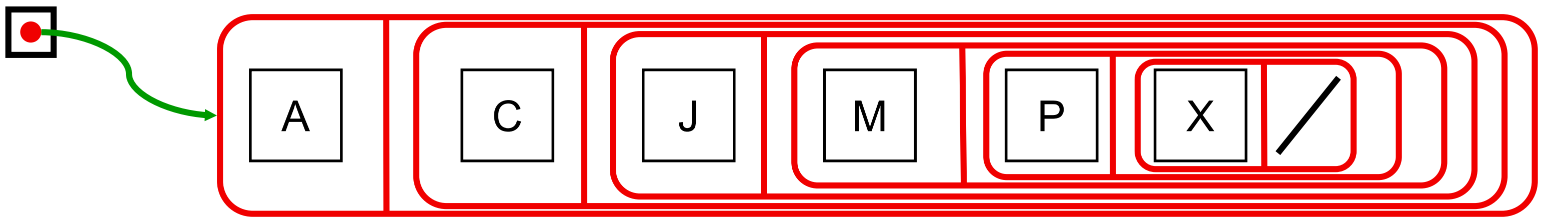
Solution ?

Delete in action

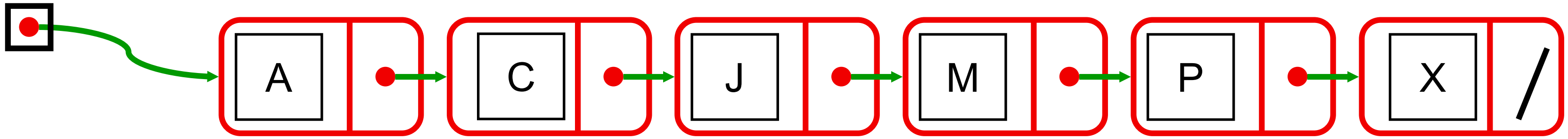


Linked List Structures – Alternative Views

- Can be drawn with Nodes inside Nodes:



- "Pointers" are better:
  - Each node contains a reference to the next node
  - reference = memory location of / pointer to object

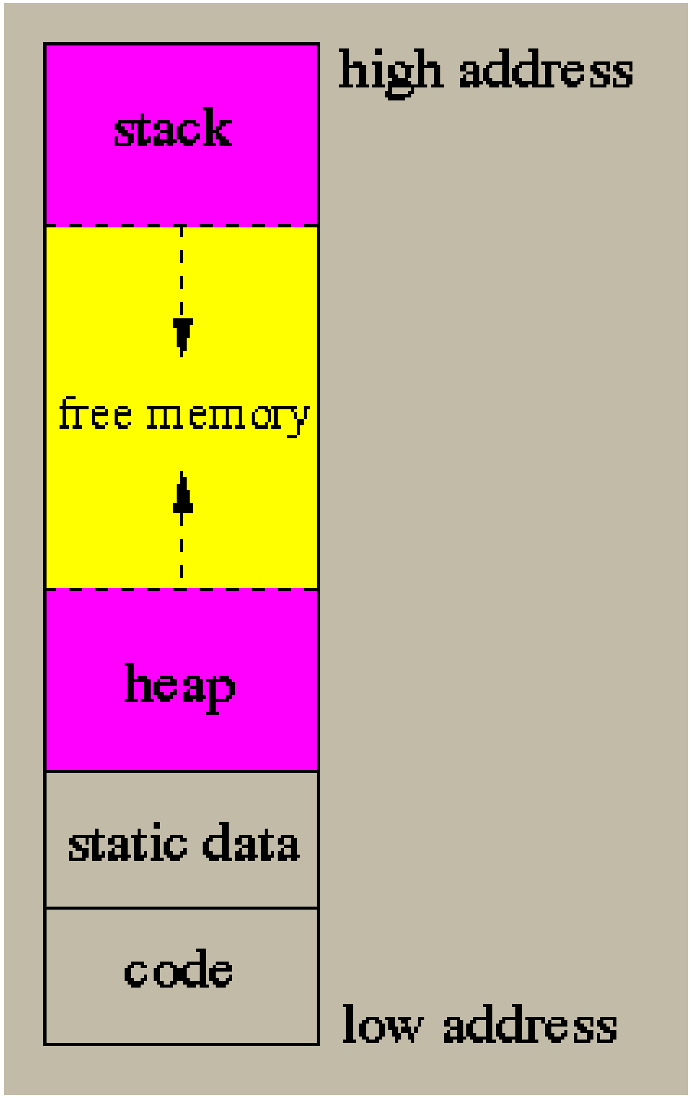


- Can view a node in two ways:
  - an object containing two fields
  - the head of a linked list of values

Aside: Memory allocation

- What are references/pointers?
  - Pointers/references are an address or a chunk of memory, where data can be stored.
- How do you get this memory allocated?
  - You've been doing it using **new**:
    - creating an object will allocate some heap memory for the object.
    - **new** returns the address of the chunk of memory
    - copying the address does not copy the chunk of memory.
- Memory from the heap must be recycled after use:
  - The **garbage collector** automatically frees up any memory chunks that no longer have anything pointing/referring to them.
  - This frees you from having to worry about explicitly freeing memory.

Heap & memory allocation

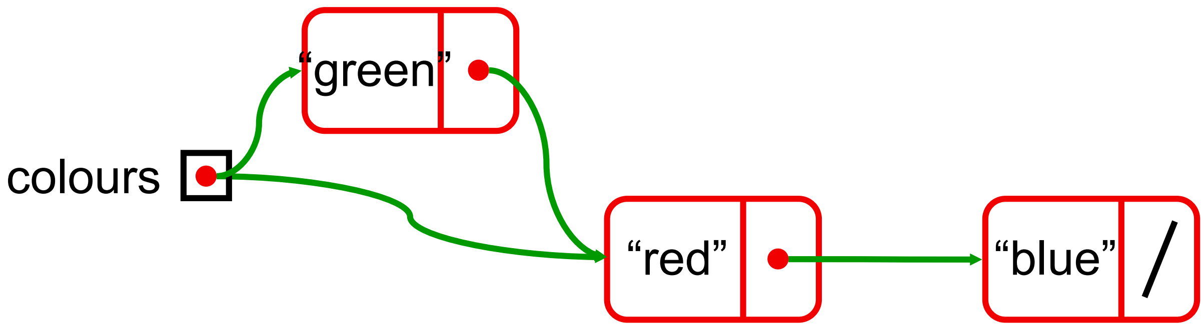


A Linked Node class:

```
public class LinkedNode <E>{
    private E value;
    private LinkedNode<E> next;
    public LinkedNode(E item, LinkedNode<E> nextNode){
        value = item;
        next = nextNode;
    }
    public E get()  {  return value;  }
    public LinkedNode<E> next()  {  return next; }
    public void set(E item)  {
        value = item;
    }
    public void setNext(LinkedNode<E> nextNode)  {
        next = nextNode;
    }
}
```

Using Linked Nodes

```
LinkedNode<String> colours = new LinkedNode<String> ("red", null);

colours.setNext(new LinkedNode<String>("blue", null));

colours = new LinkedNode<String>("green", colours);
```

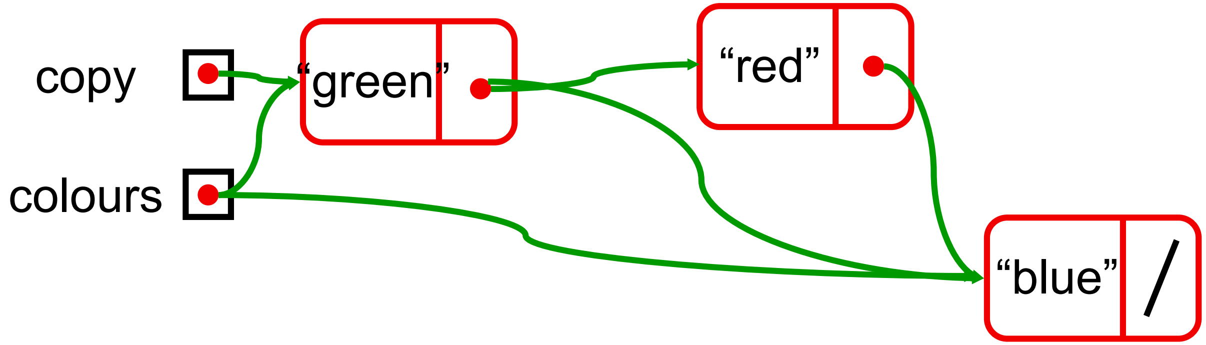



```
System.out.format("1st: %s\n",  colours.get() );                green
System.out.format("2nd: %s\n", colours.next().get() );          red
System.out.format("3rd: %s\n",  colours.next().next().get() );  blue
```

Using Linked Nodes

- Remove the second node:

```
colours.setNext(colours.next().next());
```

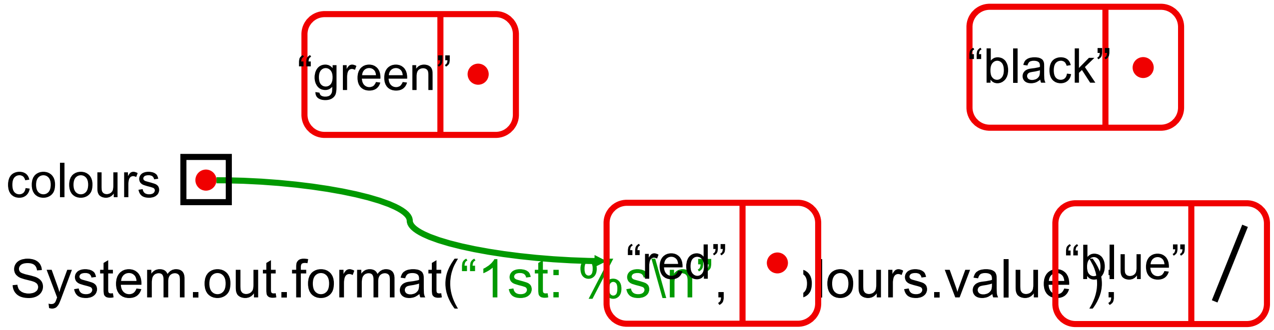


- Copy colours, then remove first node

```
LinkedNode<String> copy = colours;
colours = colours.next();
```

Using Simpler Linked Nodes

```
LinkedNode<String> colours = new LinkedNode<String> ("red", null);
colours.next = new LinkedNode<String>("blue", null);
colours = new LinkedNode<String>("green", colours);
colours.next.next = new LinkedNode<String>("black", colors.next.next);
```




```
System.out.format("1st: %s\n", colours.value );
System.out.format("2nd: %s\n", colours.next.value );
System.out.format("3rd: %s\n",  colours.next.next.value );
System.out.format("4th: %s\n",  colours.next.next.next.value );
colours.next.next.next.next = colours;
```

green

red

black

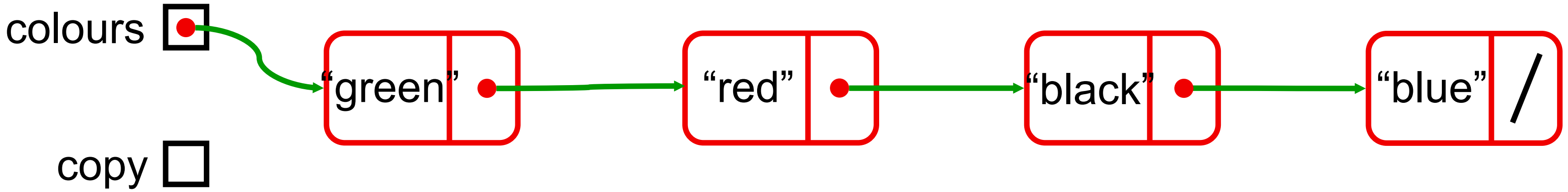
blue

A circular list: g-r-b-b-g

Using Simpler Linked Nodes

- Remove the third node:

```
colours.next.next = colours.next.next.next;
```

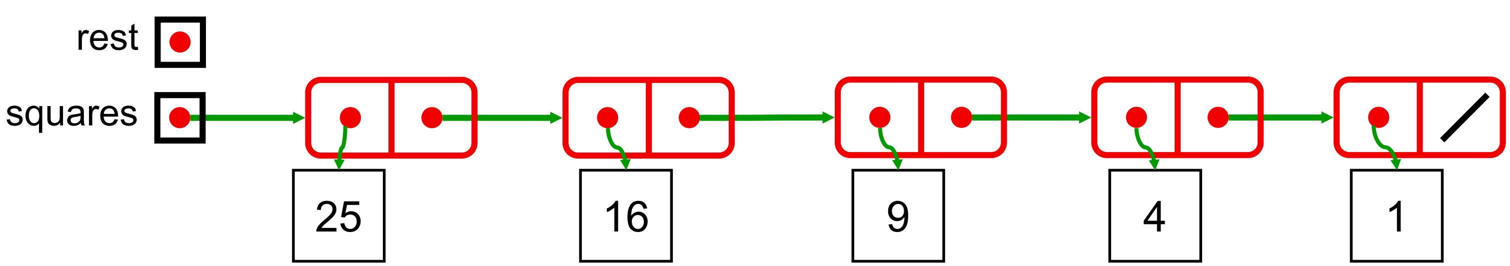


Creating & Iterating through a linked list

```
LinkedNode<Integer> squares = null;
for (int i = 1;  i < 6; i++)
    squares= new LinkedNode<Integer>( i*i, squares);

LinkedNode<Integer> rest = squares;
while (rest != null){
    System.out.format("%6d \n", rest.value);
    rest = rest.next;
}
```

or

```
for (LinkedNode<Integer> rest=squares; rest!=null; rest=rest.next){
        System.out.format("%6d \n", rest.value);
}
```



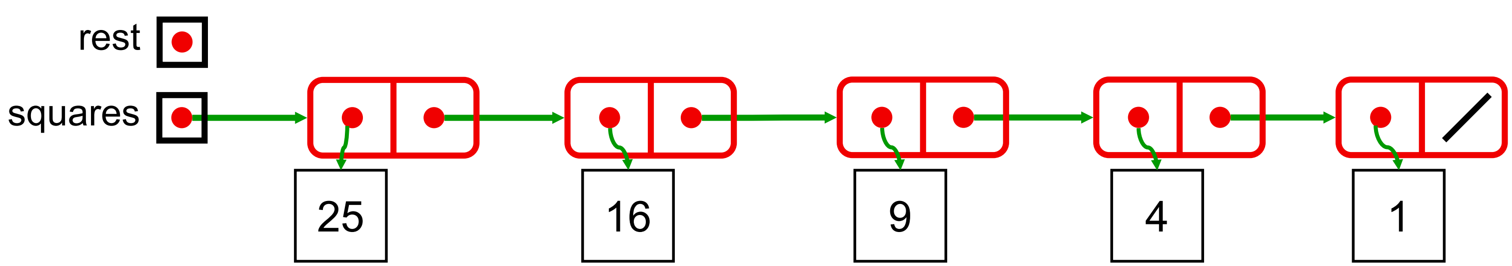
Method to print a linked list

```java
/** Prints the values in the list starting at a node */
public void printList(LinkedNode<E> list){
    if (list == null)   return;
    System.out.format("%d, ", list.value);
    printList(list.next);
}
```

or

```java
public void printList(LinkedNode<E> list){
    for (LinkedNode<Integer> rest=list;  rest!=null;   rest=rest.next )
        System.out.format("%d, ", rest.value);
}
```

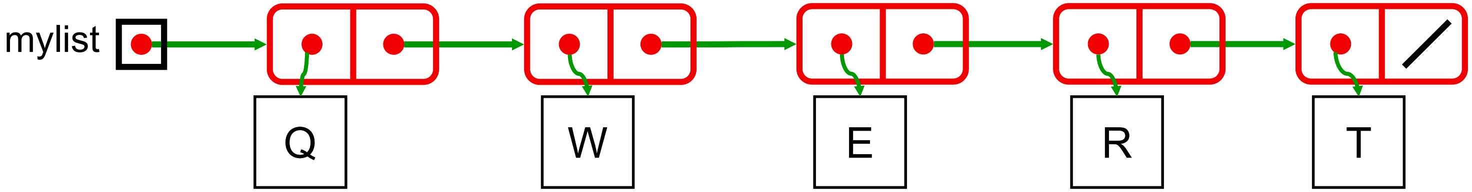
Recursive methods are generally easier to design than iterative, for recursive data structures.



Inserting:

/** Insert the value at position n in the list (counting from 0)
Assumes list is not empty, n>0, and n <= length of list */

```
public void insert (E item, int n, LinkedNode<E>list){ ….
```



Insert X at position 2 in mylist

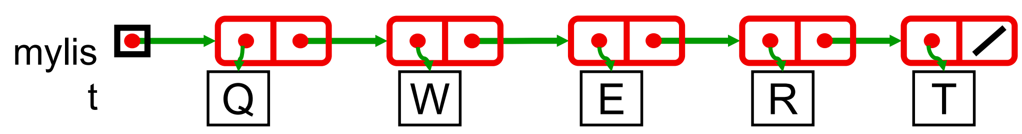
Insert Y at position 4 in mylist

Inserting:

```
/** Insert the value at position n in the list (counting from 0)
    Assumes list is not empty,  n>0, and n <= length of list */
public void insert (E item, int n, LinkedNode<E>list){
    if (n == 1 )
        list.next = new LinkedNode<E>(item, list.next);
    else
        insert(item, n-1, list.next);
}
```

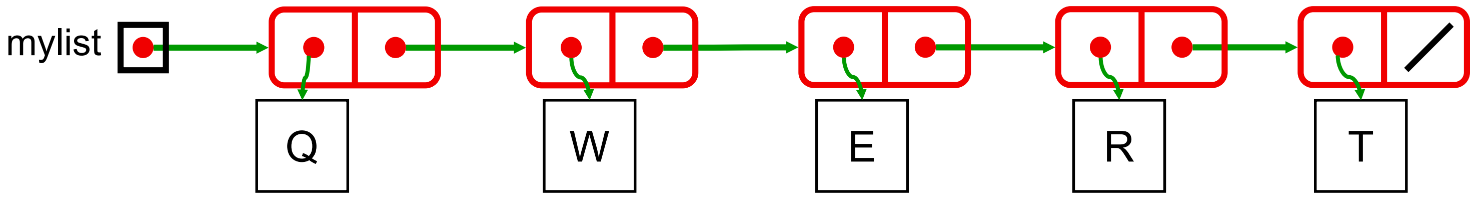
or

```
public void insert (E item, int n, LinkedNode<E>list){
    int pos =0;
    LinkedNode<E> rest=list;   // rest is the pos'th node
    while (pos <n-1){
        pos++;
        rest=rest.next;
    }
    rest.next = new LinkedNode<E>(item, rest.next);
}
```



Removing:

```
/** Remove the value from the list
    Assumes list is not empty, and value not in first node */
public void remove (E item, LinkedNode<E>list){
```



Remove R from mylist
Remove Y from mylist
Remove T from mylist

Removing:

```
/** Remove the value from the list
    Assumes list is not empty, and value not in first node */
public void remove (E item, LinkedNode<E>list){
    if (list.next==null) return;        // we are at the end of the list
    if (list.next.value.equals(item) )
        list.next =  list.next.next;
    else
        remove(item, list.next);
}
```

or

```
public void remove (E item, LinkedNode<E>list){
    LinkedNode<E> rest=list;
    while (rest.next != null  && !rest.next.value.equals(item))
        rest=rest.next;
    if (rest.next != null)
        rest.next = rest.next.next;
}
```

Why have a ‘rest’ to hold ‘list’?

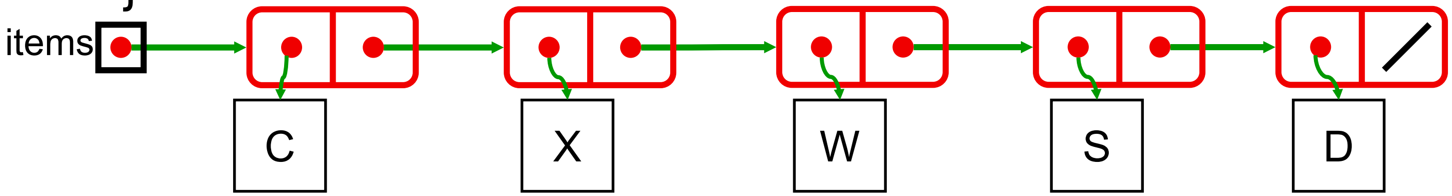
Exercise:

- Write a method to return the value in the LAST node of a list:

```
/** Returns the value in the last node of the list starting at a node */
public E lastValue (LinkedNode<E> list){  /* recursive version */




}
```

or

```
public E lastValue (LinkedNode<E> list){/* iterative version */




}
```



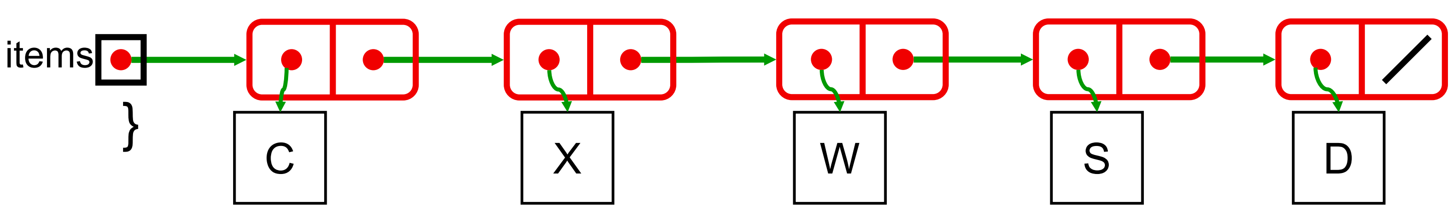
Exercise:

- Write a method to return the value in the LAST node of a list:

```
/** Returns the value in the last node of the list starting at a node */
public E lastValue (LinkedNode<E> list){



}
```

or

```
public E lastValue (LinkedNode<E> list){

}
```



Q&A

- When do you write test cases for “black box” testing? Before or after implementation?
- Explain why array implementations of queue are slow.
- Linked list allows data removal by?
- Define references/pointers.
- What is the purpose of garbage collection in memory management?

Summary

- Testing collection implementations
- Queues
- Motivation for linked lists
- Linked structures for implementing Collections

Readings

- [Mar07] Read 3.5
- [Mar13] Read 3.5