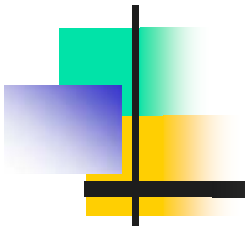
Linked lists

Insert

Delete

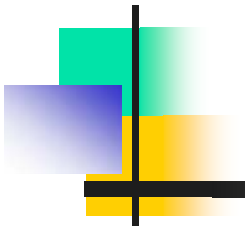
Lookup

Doubly-linked lists



Object References

- When you declare a variable of a *non-primitive* type you are really declaring a *reference* to that object
  - String foo;
  - Complex c;
- *new* actually creates the object
  - String foo = new String(“Hello”);
  - Complex c = new Complex(5, 2);
- Multiple references can refer to the same object
  - String bar = foo;
- References can be reassigned
  - foo = new String(“Goodbye”);
- *null* is a special reference that refers to nothing

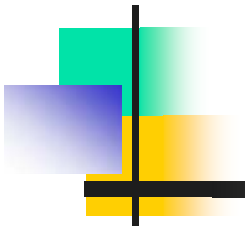


Growing an Array

- See List2 code

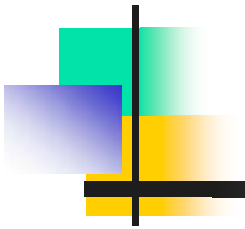
Common operation: keep a list of items

- It's extremely common in programs to want to store a collection of items
- Example: array
  - Collection of items of the same type
  - Easy to access elements of the array, but…
  - Size set at array creation time
    - Size (or max size) must be known at creation time
    - Difficult to make the array larger
    - Making the array smaller wastes space
  - Sorting requires actually moving the data around
    - Insert and delete also require copying the data



Abstract data type: list

- Common ADT: list (of objects)
- List supports several operations
  - Insert
  - Delete
  - Lookup
  - Index
  - Length
- Implementation of a list may vary
  - Array can be used to implement a list
    - Index & length are fast
    - Insert can be very slow (and waste memory)
  - Alternative: linked list



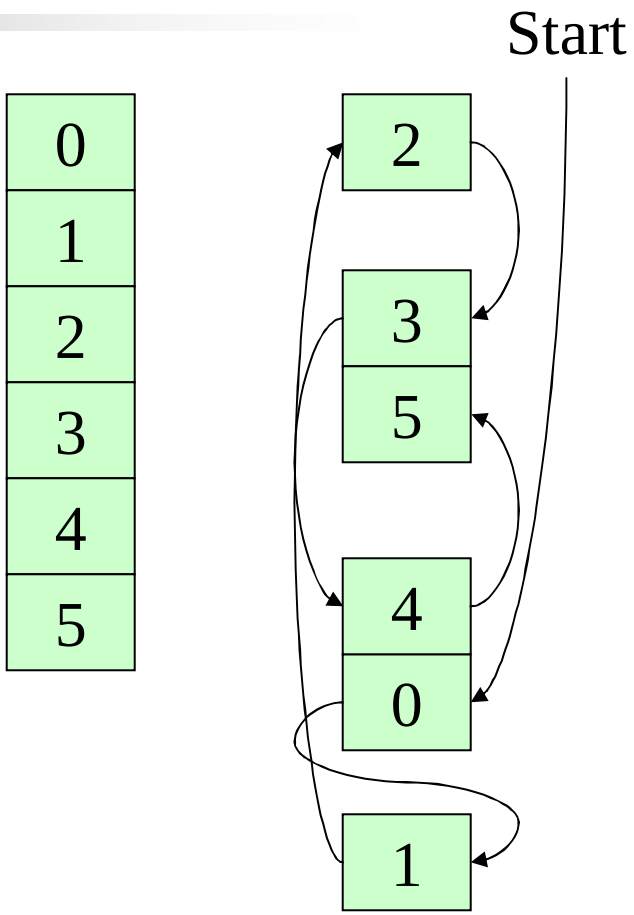
Linked Lists

- Variable length list data structure
- Each list element contains
  - Its data
  - A reference to the next element of the list
    - NULL if it is the last element of the list
- *Head* or *Start* points to the first element of the list
- Inserting an element
  - Previous element refers to new element, new element refers to the one the previous element used to refer to
- Removing an element
  - Previous element refers to whatever the removed element used to refer to

Linked lists versus arrays

- Array: objects occupy contiguous memory

- Linked list: objects need not be contiguous
  - Each object refers to others in the list
  - Singly-linked list: each object has a reference to the next one in the list
  - Doubly-linked list: each object has a reference to both the previous and next ones in the list



So how is this done in Java?

- Java: reference is a *name* for an object
  - One object can have multiple names
  - Changes in the object seen through all names
- Two types make up a list
  - *Header* type: used for the list itself
  - *Node* type: used for elements in the list
    - Object being stored: may store reference or new object
    - Reference to other nodes

```
public class SLList {
  SLLNode start;
  int count;
}

class SLLNode {
  SLLNode next;
  Object obj;
}
```

Why use Object in the list?

- Lists contain values
  - Strings
  - Numbers
  - More complex data structures
- We want to build a list that'll work with anything!
  - Write the code once and reuse it!
- All types in Java except builtins are descended from Object
  - Builtins like int must use provided classes like Integer
  - List can now be used for anything!

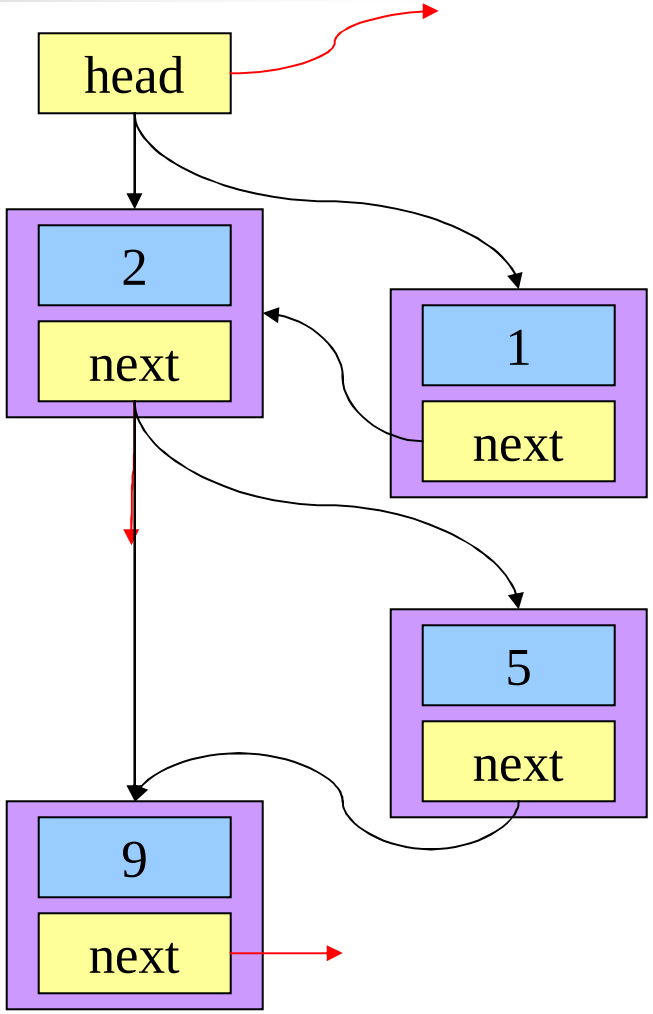
Definitions for SLList and SLLNode

```
public class SLList {
    private SLLNode start;
    private int count;
    public void add (int index, Object item) throws ArrayIndexOutOfBoundsException;
    public void remove (int index) throws ArrayIndexOutOfBoundsException;
    public Object get (int index) throws ArrayIndexOutOfBoundsException;
    private SLLNode getIndex (int index) throws ArrayIndexOutOfBoundsException;
    public void removeAll();

    private class SLLNode {
        SLLNode next;
        Object obj;
    }
}
```

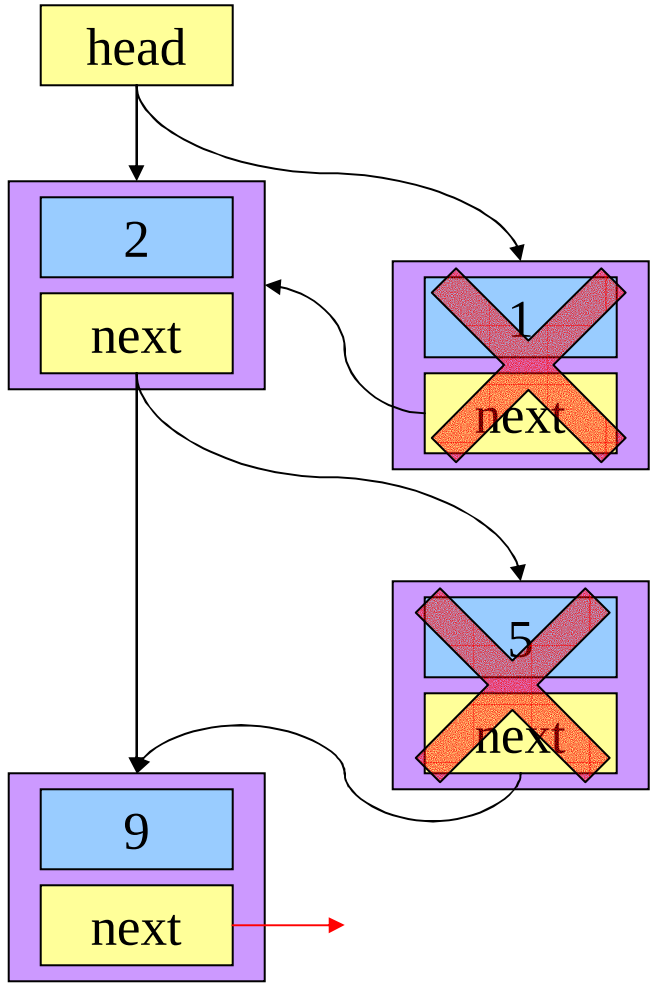
Inserting into a singly linked list

- Inserting into a link list has two cases
  - First in the list
  - Not first in the list
- If going at head, modify head reference (only)
- If going elsewhere, need reference to node before insertion point
  - New node.next = cur node.next
  - Cur node.next = ref to new node
  - Must be done in this order!



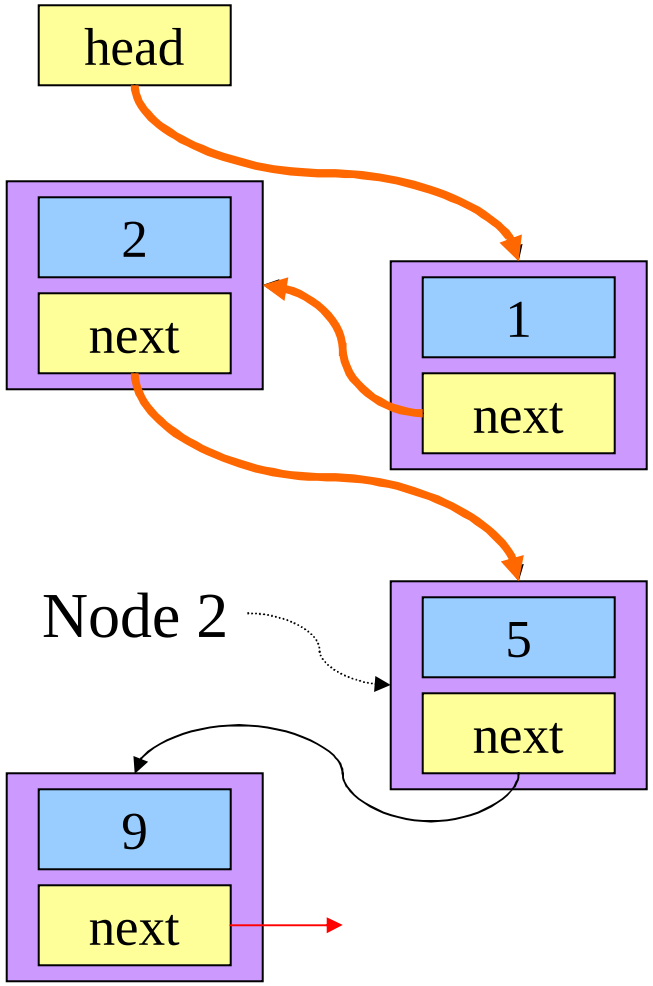
Deleting from a singly linked list

- Deleting a link list has two cases
  - First in the list
  - Not first in the list
- If deleting from head, modify head reference (only)
- If deleting elsewhere, simply “point around” the deleted node
- Space for deleted nodes is automatically reclaimed (garbage collection)



Traversing a singly linked list

- Start at the head
  - Use a "current" reference for the current node
- Use the "next" reference to find the next node in the list
- Repeat this to find the desired node
  - *N* times to find the *n*th node
  - Until the object matches if looking for a particular object
    - Caution: objects can "match" even if the references aren't the same…
- Don't forget to check to see if this is the last node

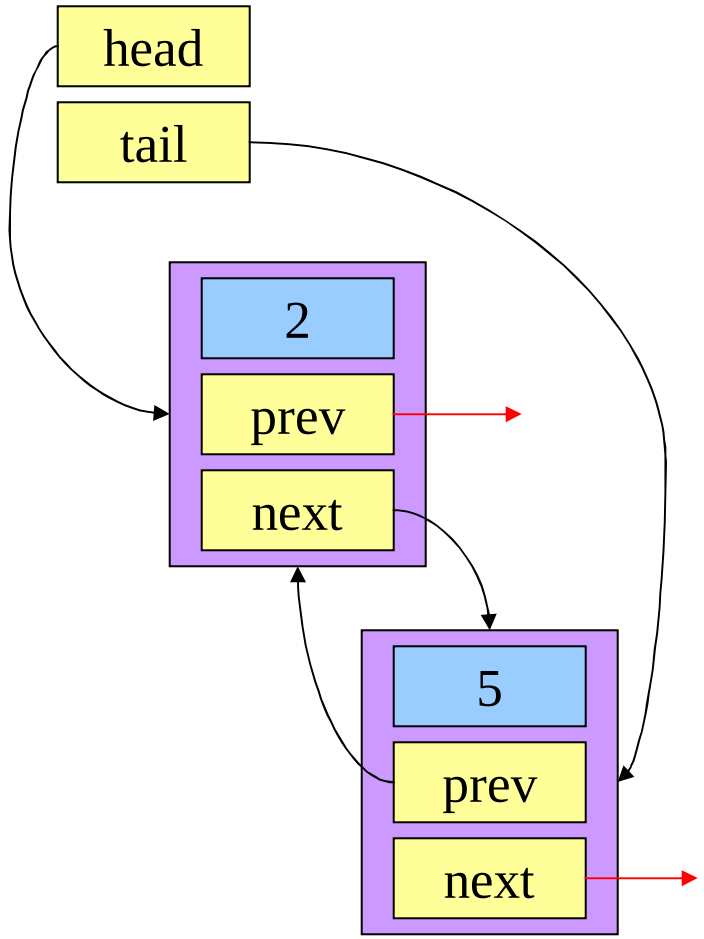


More on traversing singly linked lists

- Check for "end of list" *before* moving on to the next element!
  - Trying to dereference a null reference generates an exception
  - Java (and C) ignore statements in logical operations if they're not "needed"
    - Example
      ```
      while (n != null && n.obj != whatWeWant) {
      n = n.next
      }
      ```
    - If n is *null*, the second half of the expression is *not* evaluated
- Be careful with object comparisons!
  - String s = "hello";
  - String t = "hello";
  - At this point, s is *not* equal to t: (s == t) results in **false**
    - **Reason: s and t refer to different objects with the same content**
  - Solution: use String.equals or similar method to compare object contents rather than references

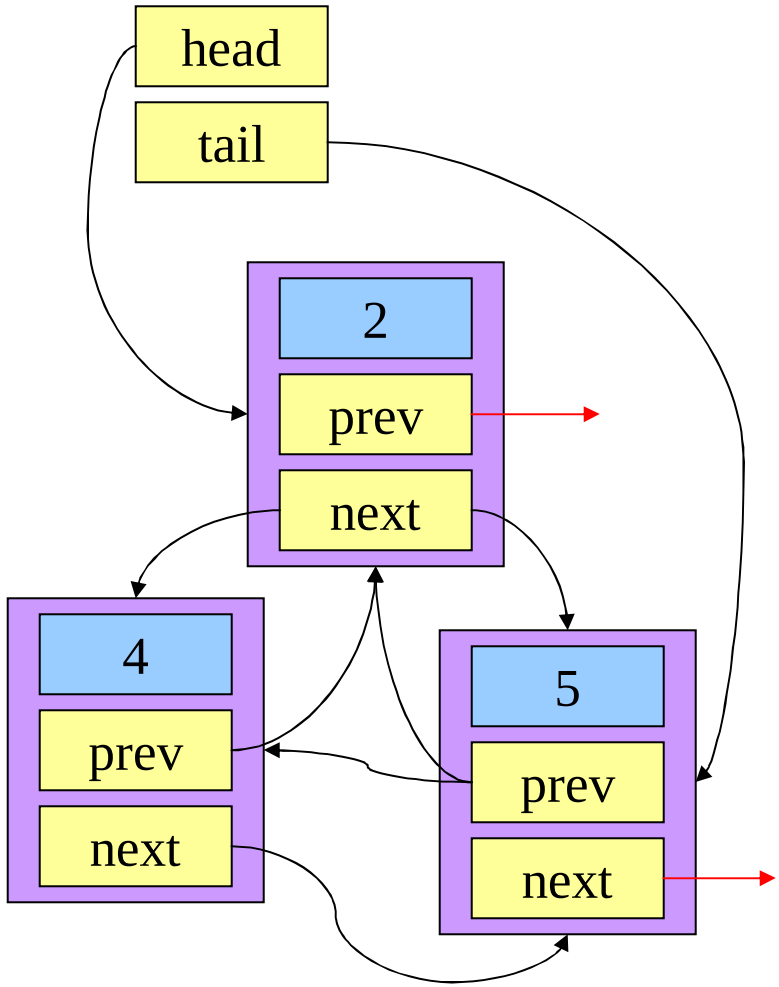
Doubly-linked lists

- Each node in the list refers to its predecessor as well as successor
  - Twice as many references
  - Easier to insert / delete nodes
  - List can be traversed in either direction
- List typically has both head and tail references
- Insertion only needs a reference to an adjacent node
- Deletion only needs a reference to the node being deleted



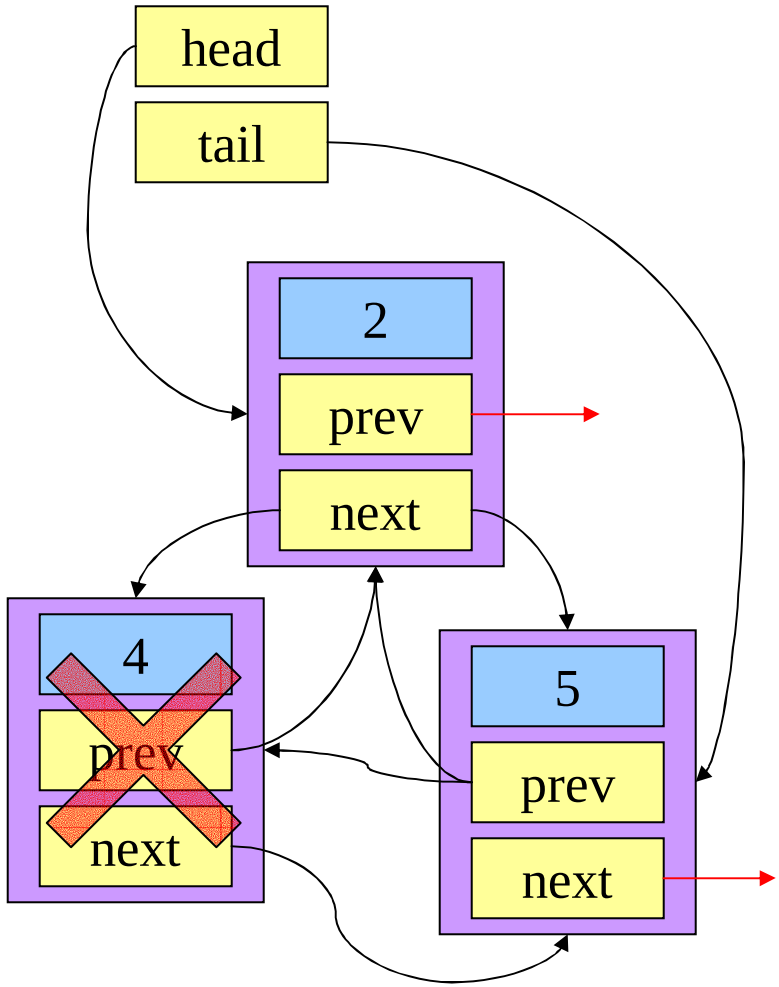
Inserting into a doubly-linked list

- As with singly linked lists, special case for head
  - Also special case for tail
- Need to update two nodes
  - Node before new node
  - Node after new node
- Hook up new node *before* modifying other nodes
  - Don't overwrite necessary information before relocating it!
- Head & tail: if a link is null, update the head or tail as appropriate



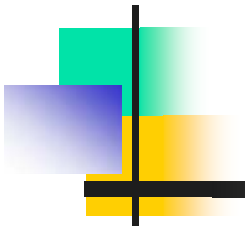
Deleting from a doubly-linked list

- As with singly linked lists, special case for head
  - Also special case for tail
- Need to update two nodes
  - Node before new node
  - Node after new node
- Hook up new node *before* modifying other nodes
  - Don't overwrite necessary information before relocating it!
- Head & tail: if a link is null, update the head or tail as appropriate



Summary of list operations

- Insert objects
  - May insert at any position
  - Special methods to insert at head or tail?
- Delete objects
  - Delete by index
  - Delete by content
- Find objects
  - Find by index number
  - Find by (incomplete?) content
- Find the length of the list



Other

- Let’s write a linked list class
- Tail reference
- Circular linked list
- Circular doubly linked list
- Dummy head node