COMP 202
The Linked List

CONTENTS:

- Aliases as pointers
- Self-referencing objects
- Abstract Data Types

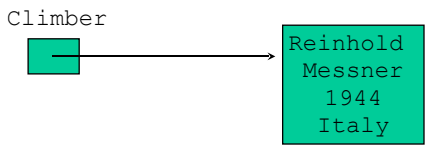
Thinking Like A Programmer:
When to use Dynamic Programming

Static vs. Dynamic Structures

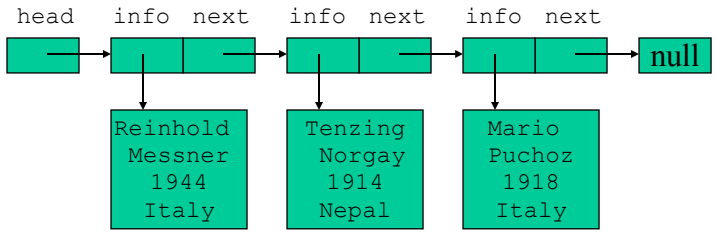
- A *fixed* data structure has a fixed size
  - Arrays: once you define the number of elements it can hold, this number can't be changed anymore
- A *dynamic* data structure grows and shrinks as required by the information it contains

Object References

- Recall that an *object reference* is a variable that stores the address of an object
- A reference can also be called a *pointer*
- They are often depicted graphically:

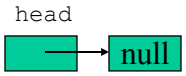


Linked List of Climbers

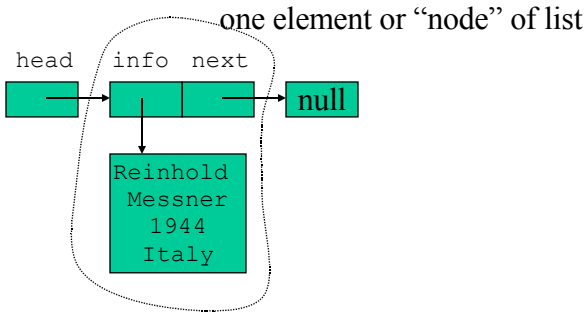


Initializations

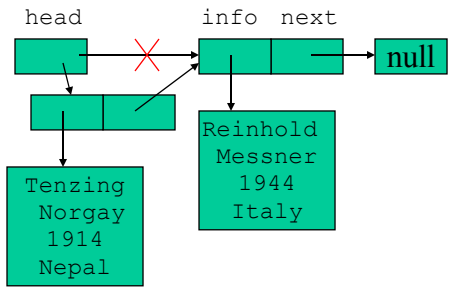
- Empty List



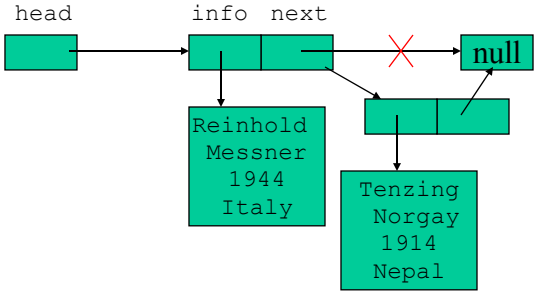
- List with one climber



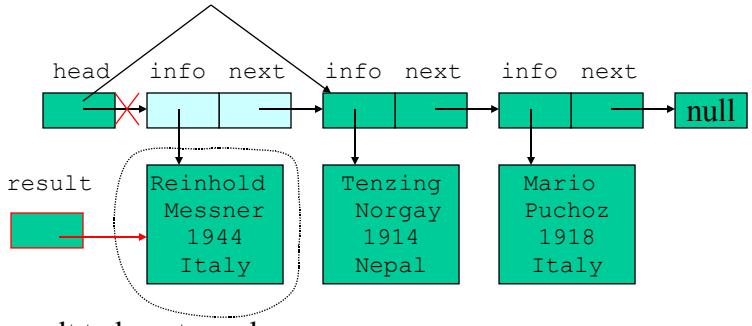
Adding a node at front



Adding a node at end



Return first node and remove it from list



result to be returned

Return a node with certain property and remove it from list

- Example: Remove climber Tenzing Norgay and return it
- go through list
  - for each node
    - if info points to the specific climber
      - adjust pointers to remove climber
      - return climber
  - once at end, return empty climber

Climber Example

```
public class Climber
{
   private String name;
   private int birthYear;
   private String nationality;

   public Climber (String name, int birth, String country)
   {
      this.name = name;
      birthYear = birth;
      nationality = country;
   }

   public String toString ()
   {
      return name + ", born in " + birthYear + " in " + nationality;
   }

   public boolean equals (String name)
   {
     return (this.name).equals(name);
   }
}
```


Climber Example

```
class ClimberNode {
   public Climber info;     // points to climber of this node
   public ClimberNode next; // points to next node

   public ClimberNode (Climber climber) {
     info = climber;
     next = null;
   }
}
```


1 of 3

```
public class ClimberList {
   private ClimberNode head;

   // constructor: create empty list
   public ClimberList() { head = null; }

   // add to front
   public void addFront (Climber newClimber) {
      ClimberNode node = new ClimberNode (newClimber);
      node.next = head;
      head = node;
   }

   // add to end
   public void addEnd (Climber newClimber) {
      ClimberNode node = new ClimberNode(newClimber);
      // pointer to a node in list
      ClimberNode current;
      // list is empty; this is the first node to enter
      if (head == null) head = node;
      else {
         current = head; // go through the list until end
         while (current.next != null)
            current = current.next; // move forward
         current.next = node; // make node the last node of list
      }
   }
```

2 of 3

```
public Climber remove (String climberName) {
   ClimberNode current = head;  // initialize pointer
   ClimberNode previous = null; // track last position of pointer
   Climber result = null;

   if (current == null) // empty list
      return result;

   do {
      if (current.info.equals(climberName)) { // climber found
         // found the climber
         result = current.climber;
         if (previous==null) // slightly different if first climber
            head = current.next;  // reassign head pointer
         else
            previous.next = current.next; // reassign ptr of previous
         return result;
      } else { // current one is not the climber
         previous = current; // the current becomes previous
         current = current.next; // move forward to the next
      }
   } while (current != null);
   // no climber with name found
   return result;
}
```

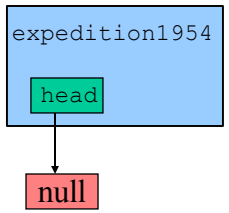

3 of 3

```
   public String toString () {
      String result = "";
      ClimberNode current = head;
      while (current != null) {
         result += current.climber.toString() + "\n";
         current = current.next;
      }
      return result;
   }

   public Climber removeFirst() {
      Climber result;
      if (head == null)
         result = null;
      else {
         result = head.info;
         head = head.next;
      }
      return result;
   }
}
```

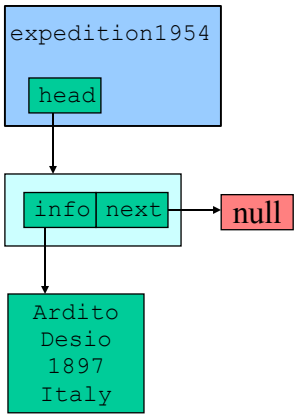

Example Main

```
public class K2Ascent {
   public static void main (String[] args) {
      ClimberList expedition1954 = new ClimberList();

      expedition1954.addFront (new Climber("Ardito Desio",1897,"Italy"));
      expedition1954.addFront (new Climber("Mario Puchoz",1918,"Italy"));
      expedition1954.addFront (new Climber("Lino Lacedelli",1925,"Italy"));
      expedition1954.addFront (new Climber("Achille Compagnoni",1914,
                                           "Italy"));

      System.out.println("Original team to attempt the first successful ascent of K2:\n");
      System.out.println (expedition1954);


      expedition1954.remove ("Ardito Desio");
      expedition1954.remove ("Mario Puchoz");


      System.out.println("\nAscenders to summit K2 (8611m):\n");
      System.out.println (expedition1954);
   }
}
```

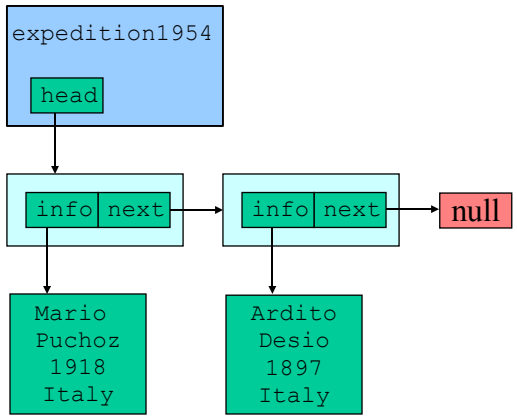

Example



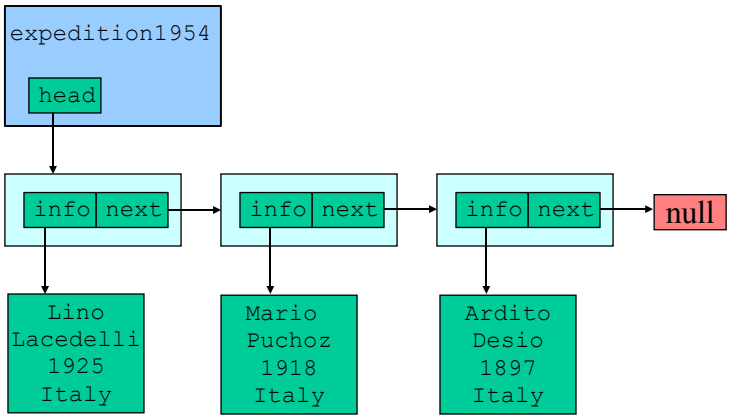
Example



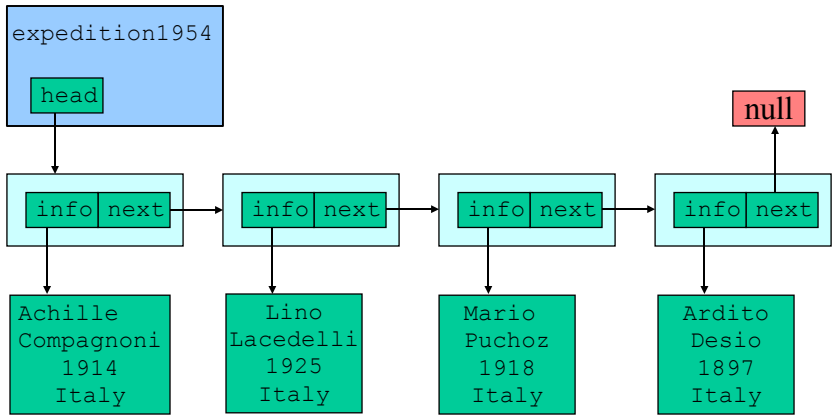
Example



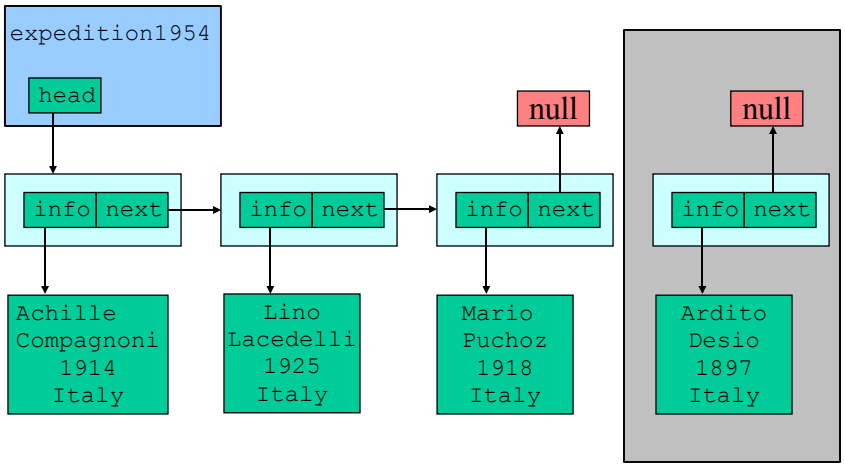
Example



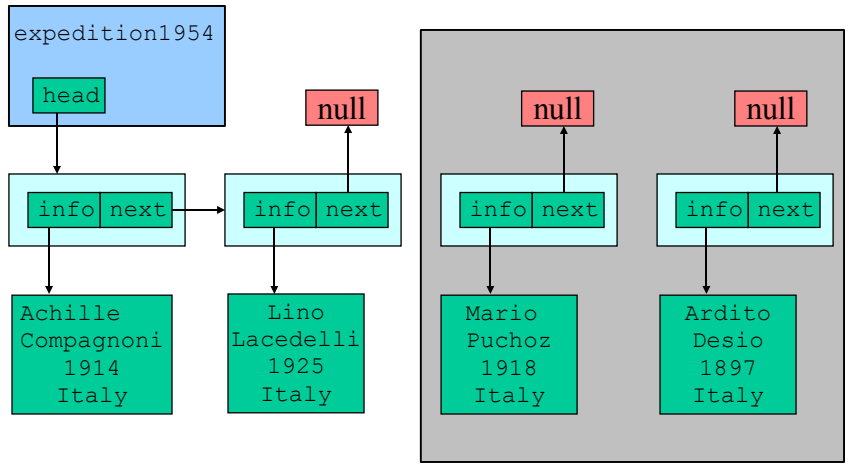
Example



Example



Example



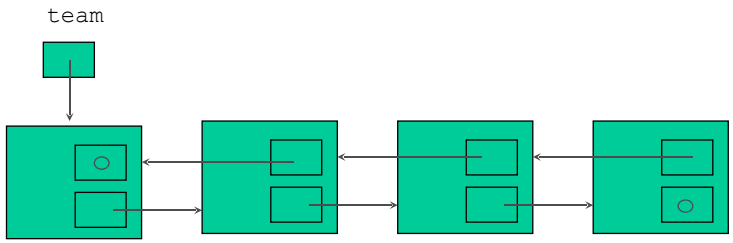
Abstract Data Type

- Our climber list is kind of an abstract data type
- It provides list functionality
  - add at front
  - add at end
  - remove
  - …
- The user has a useful way of collecting data
- The user does not need to know how the methods are actually implemented
  - ClimberList could have also used an ArrayList…

Other Dynamic Structures (Only to be aware of – not to code)

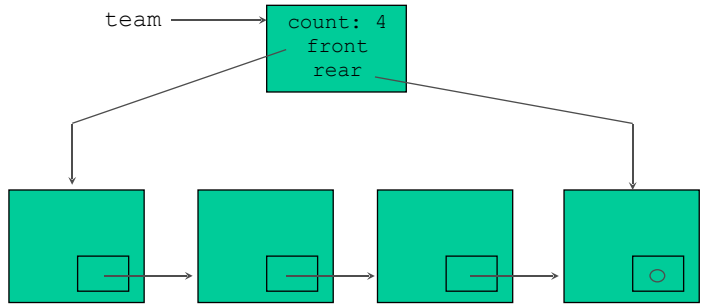
Other Dynamic List Implementations

- It may be convenient to implement as list as a *doubly linked list*, with next and previous references:



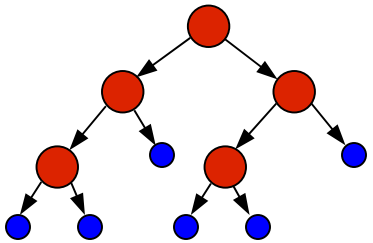
Other Dynamic List Implementations

- It may also be convenient to use a separate header node, with references to both the front and rear of the list

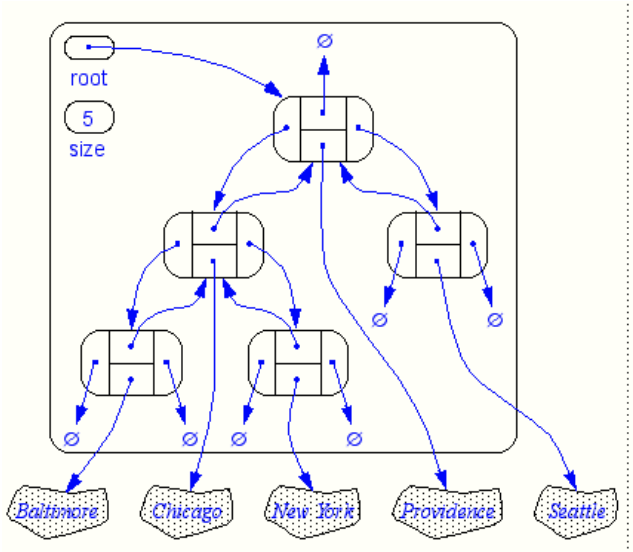


Trees

- A *tree* is a data structure that represents a hierarchy, through internal and external nodes
- Ex: table of contents for a book, OS file system, inheritance relationship between Java classes, organizational structure of a corporation, etc.
- A *binary tree* is a tree where each internal node has exactly 2 child nodes. A Binary tree is either (recursive definition):
  - An external node (a *leaf*)
  - An internal node and two binary trees (left subtree and right subtree)

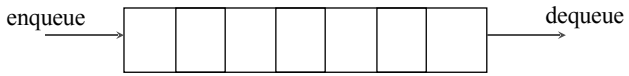


Linked Tree Implementation



Queues

- A *queue* is similar to a list but adds items only to the end of the list and removes them from the front
- It is called a FIFO data structure: First-In, First-Out
- Analogy: a line of people at a bank teller's window



Queues

- We can define the operations on a queue as follows:
  - enqueue - add an item to the rear of the queue
  - dequeue - remove an item from the front of the queue
  - empty - returns true if the queue is empty
- As with our linked list example, by storing generic `Object` references, any object can be stored in the queue
- Queues are often helpful in simulations and any processing in which items get "backed up"

Part 4

Thinking like a programmer

When to use dynamic structures…

- We have two kinds of structures in computers:
  - Structures that have a predefined size and never change (called fixed structures)
  - Structures that can be built (or re-formed) at run-time (called dynamic structures)
- Generally speaking dynamic data is slower to execute than fixed data
  - If you can get away with using fixed structures then do so.
  - If you do not know how big your structure should be or if it requirements change while the program executes then use dynamic structures.

Designing For Dynamic Structures

- First define the structure of the node
  - need a self-referencing pointer to link to other nodes
- Make sure there exists at least one header reference that points to the beginning of your structure
- Now, determine where in your code the structure should be:
  - Assembled
  - Disassembled
  - Restructured
- Now create a class that will manage that using encapsulation