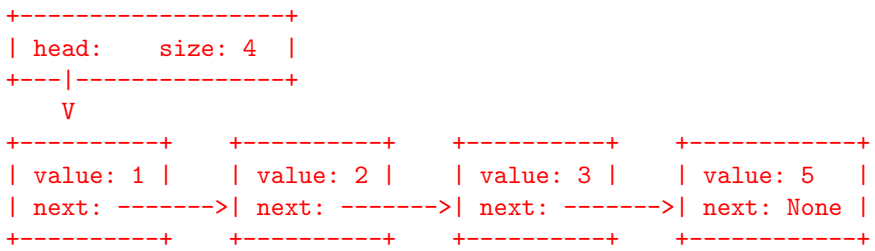
Linked Lists

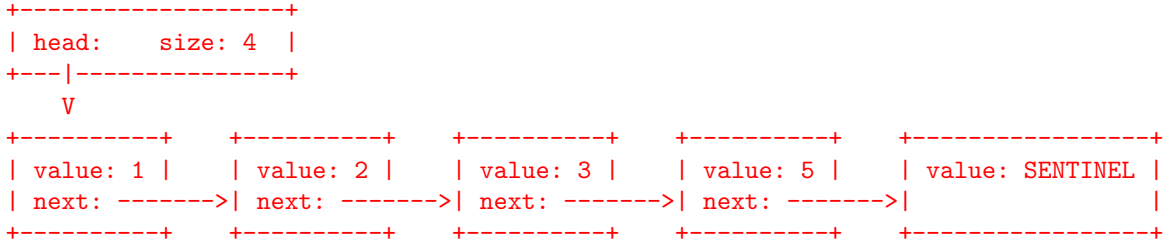
1. You are given the linked list: $1 \rightarrow 2 \rightarrow 3$. You may assume that each node has one field called `value` and one called `next`.

   (a) 1 points to 2 and 2 points to 3. What does 3 point to?
   The implementer may cause the 3 node to point to either `None` or a sentinel node.

   (b) Draw out the linked list structure and add a 5 to the end.



OR



   (c) Write pseudocode to add an element onto the end of a linked list.

```
def append(linked_list, value):
    newEnd = new node
    newEnd.value = value
    newEnd.next = None # (or SENTINEL)
    cur = linked_list.head
    while cur is not end of linked_list:
        cur = cur.next
    cur.next = newend
```

(d) Let's say we want to add a 4 to the list from 1b. What is a procedure for inserting this value at a certain position in the linked list? (There are many possibilities.)
Use some variation of this strategy:
Find the preceding element's node, link its next node to the new node containing 4, then link that new node to what was the following node.

Binary Search using Python Lists

2. Given the sorted list [1, 4, 9, 16, 25, 69, 420, 1337], write out the steps that a binary search would take to find the number 69.

[1,4,9,16,|25|,69,420,1337]
[1,4,9,16,25,69,|420|,1337]
[1,4,9,16,25,|69|,420,1337]

Naive Sorting

3. Below is Python code for a function that performs an insertion sort and prints `data` after each iteration of the `for` loop.

```
1 def insertion_sort( data ):
2     for marker in range( 1, len( data ) ):
3         val = data[marker]
4         i = marker
5         while i > 0 and data[i-1] > val:
6             data[i] = data[i-1]
7             i -= 1
8         data[i] = val
9         print( data )
```

(a) Write out what the function will print for the input list: [3,2,7,1].

[2,3,7,1]
[2,3,7,1]
[1,2,3,7]

(b) What is the sort algorithm's time complexity?
$O(N^2)$

Stacks and Queues

4. Suppose we wanted to implement a stack by using a linked list. Define linked-list operations that would correspond to `push()` and `pop()`.

   `push( stack, element )` → `insertFront( linked-list, element )`
   `pop( stack )` → `removeFront( linked-list )`

5. It is possible to implement both stacks and queues using only simple Python lists.

   (a) Write the following functions that implement stack functionality atop a Python list. Your stack must provide the following functionality:

   `push(lst, elm)` — Push `elm` onto the top of the stack

   `pop(lst)` — Return the top element of the stack and remove it from the stack

   `isEmpty(lst)` — Return whether the stack is empty

   `peek(lst)` — Return the top element of the stack without modifying the stack

```
1  def push(lst, val):
2      lst.append(val)
3  def pop(lst):
4      return lst.pop()
5  def peek(lst):
6      return lst[-1]
7  def isEmpty(lst):
8      return (len(lst) == 0)
```

   (b) Write the following methods to create a queue in similar fashion to previous question (with a Python list as the data structure managing elements "under the hood"):

   `enqueue(lst, val)` - put a value into the queue

   `dequeue(lst)` - take a value out of the queue

```
1  def enqueue(lst, val):
2      lst.append(val)
3  def dequeue(lst):
4      lst.pop(0)
```

   (c) Which of the data structures you implemented is more efficient and why? Give a better way to implement the slower structure, and discuss how this would change the time complexity of its operations.
   Because the queue must be able to modify both ends of the list, it pays an O(n) cost to remove the beginning element during each dequeue operation. This could be reduced to O(1) by using a linked list instead of a Python list.

Hashing and Hash Tables

6. Chris made a mistake in his hash table implementation!

```
1  from rit_lib import *
2
3  class WonkyHashTable( struct ):
4      _slots = ( (int,"size"), (list,"table") )
5
6  def createWonkyHashTable(size=5):
7      table = [None for i in range(size)]
8      return WonkyHashTable(size, table)
9
10 def add_element(wonkyHashTable, element):
11     hash = bad_hash(wonkyHashTable, element)
12     wonkyHashTable.table[hash] = element
13
14 def contains(wonkyHashTable, element):
15     return element in wonkyHashTable.table
16
17 def bad_hash(wonkyHashTable, element):
18     return len(element) % len(wonkyHashTable.table)
19
20 # main program
21
22 htable = createWonkyHashTable()
23 for elm in 'I wrestled a bear once'.split():
24     add_element(htable, elm)
```

(a) Show what the hash table looks like after the for loop on line 23 completes.

`[None, 'a', None, 'wrestled', 'once']`

(b) What is wrong with the code? What can we do to make the function behave as Chris expects it to behave?

The issue is on line 12. Whenever `bad_hash()` dictates that an element should be placed in an occupied bucket, that bucket's contents get overwritten! Change

```
wonkyHashTable.table[hash] = element
```

to

```
while wonkyHashTable.table[hash] is not None:
    hash = (hash + 1) % wonkyHashTable.size
wonkyHashTable.table[hash] = element
```

Note that this code assumes that the specified `size` of the table is large enough to hold all elements. This is usually not the case.

(c) Draw the table of the properly behaving hash function.

`['once', 'I', 'a', 'wrestled', 'bear']`

(d) Assuming that this hash table will only be used on strings, is the hashing function being used a good one? Why or why not?

No: It ignores the fact that most English words are the roughly the same length. The number of collisions is expected to be massive. We should take advantage of the characters in the input strings, not the number of characters.

Structures

7. For each of the following structures with the given attributes, write the structure code and a corresponding maker function:

   - A Hotel
     - `name` (a string)
     - `rooms` (a list of Room structures)
     - `location` (a string)
   - A Room
     - `number` (an integer)
     - `capacity` (an integer)
     - `price` per night (a float)

```
1 from rit_lib import *
2
3 class Hotel( struct ):
4     _slots = ( (str,"name"), (list,"rooms"), (str,"location") )
5
6 def createHotel(name, rooms, location):
7     return Hotel(name, rooms, location)
8
9 class Room( struct ):
10    _slots = ( (int,"number"), (int,"capacity"), (float,"price") )
11
12 def createRoom(number, capacity, price):
13    return Room(number, capacity, price)
```

Greedy Algorithms

8. Given that an algorithm is *greedy*, is it guaranteed to return an *optimal* solution?

   NO. Greedy algorithms always choose the *current* best solution, which is not necessarily the *overall* best solution!

9. In the game Black and White[1], the player is faced with a row of identical double-sided chips. You can probably guess what colors the two sides of each chip are. The objective is to flip as many chips as necessary so their exposed colors match that of a target pattern.
   The catch? Reordering the chips is said to be Impossible by those who seem to know what they're talking about.
   The *real* catch? Flipping a group of consecutive tiles can be accomplished in a single "action." If every flip takes one "action," write a function `bwMoves` that, given a starting pattern and target pattern as equal-length strings, returns the minimum number of actions required to get them to match. For instance, `bwMoves( 'BBWBBWBBBB', 'WWWWWBBWWB' )` should return 3.

```
1  def bwMoves(start, target):
2      actions=0
3      first=0
4      for index in range(len(start)):
5          if start[index]==target[index]:
6              if first!=index: # Each flip works up to (but not including)
7                  actions+=1   # the index pointer. If first==index, that's
8              first=index+1    # 0 elements, so there is nothing to flip.
9                               # (i.e. There were two no-flips in a row.)
10     if start[-1]!=target[-1]:
11         actions+=1
12     return actions
```

[1] Special thanks to Professor Butler for unwittingly allowing us to rip off his problem.