CS106B Chris Gregg
Spring 2018

Section Handout #5 Solutions

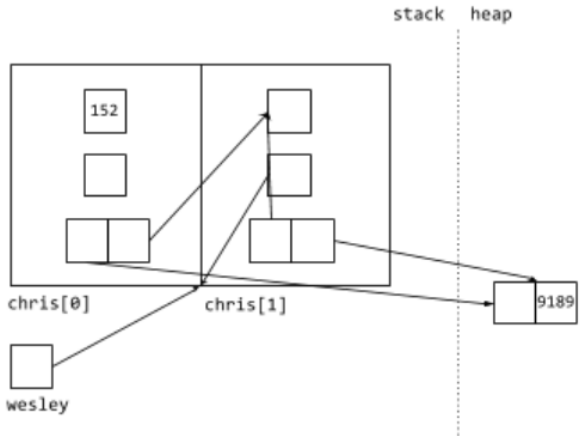
If you have any questions about the solutions to the problems in this handout, feel free to reach out to your section leader, Nick, or Chris for more information.

1. A Series of Unfortunate References

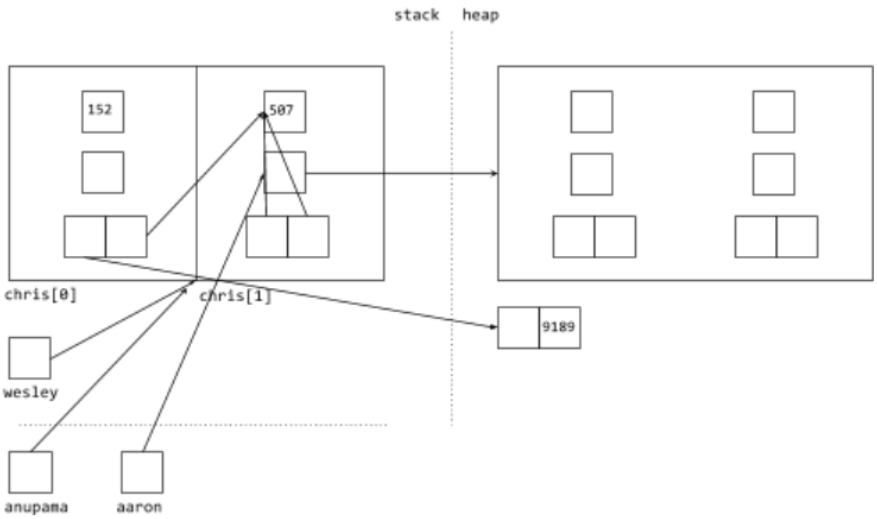
```
1 -84 2
```

2. Section Leaders, Then and Now

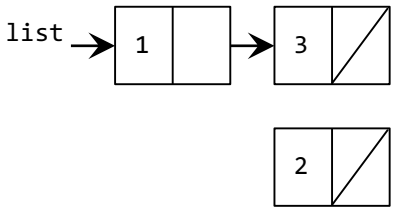
Below is the state of memory just prior to the call to `tyler`:

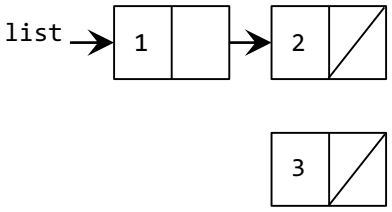


Below is the state of memory just before the call to `tyler` exits:



3. What's the Code Do?





4. What's the Code?

```
// a)
ListNode *temp = list->next->next;
temp->next = list->next;
list->next->next = list;
list->next->next->next = NULL;
list = temp;
```

```
// b)
list->next->next->next = list;
list = list->next->next;
ListNode *list2 = list->next->next;
list->next->next = NULL;
```

5. Is Sorted

```
bool isSorted(ListNode *front) {
  if (front != NULL) {
    ListNode* current = front;
    while (current->next != NULL) {
      if (current->data > current->next->data) {
        return false;
      }
      current = current->next;
    }
  }
  return true;
}
```

6. Count Duplicate Strings

```
int countDuplicateStrings(StringNode *front) {
  int count = 0;
  if (front != NULL) {
    StringNode *current = front;
    while (current->next != NULL) {
      if (current->data ==current->next->data) {
        count++;
      }
      current = current->next;
    }
  }
  return count;
}
```

7. Remove All Threshold

```
void removeAllThreshold(DoubleNode *&front, double value, double threshold) {
  while (front != NULL && front->data >= value - threshold
                       && front->data <= value + threshold) {
    DoubleNode *trash = front;
    front = front->next;
    delete trash;
  }
  if (front != NULL) {
    DoubleNode *current = front;
    while (current->next != NULL) {
      if (current->next->data >= value - threshold
       && current->next->data <= value + threshold) {
        DoubleNode *trash = current->next;
        current->next = current->next->next;
        delete trash;
      } else {
        current = current->next;
      }
    }
  }
}
```

8. Double List

```
void doubleList(ListNode *&front) {
  if (front != NULL) {
    ListNode *half2 = new ListNode(front->data);
    ListNode *back = half2;
    ListNode *current = front;
    while (current->next != NULL) {
      current = current->next;
      back->next = new ListNode(current->data);
      back = back->next;
    }
    current->next = half2;
  }
}
```

9. Split

```
void split(ListNode *&front) {
    if (front != nullptr) {
     // Points to the latest negative number found
       ListNode *lastNegative = nullptr;
       if (front->data < 0) {
          lastNegative = front;
       }

       // Operate one element ahead in case we need to rewire
       ListNode *current = front;
       while (current->next != nullptr) {
           if (current->next->data < 0) {
                // Have current element reroute around this element
               ListNode *temp = current->next;
               current->next = current->next->next;
               if (lastNegative == nullptr) {
               // If first negative we've seen, it should be at front
                   lastNegative = temp;
                   lastNegative->next = front;
                   front = lastNegative;
```

```
                } else {
                // This should be at the end of the negatives list
                    temp->next = lastNegative->next;
                    lastNegative->next = temp;
                    lastNegative = temp;
                    // Jump ahead if this element is still next
                    // (could happen if front is negative)
                    if (current->next == lastNegative) {
                        current = current->next;
                    }
                }
            } else {
                current = current->next;
            }
        }
    }
}
```

10. Reverse Recurse

```
ListNode *reverse(ListNode *front) {
  if (front == NULL) return NULL;
  if (front->next == NULL) return front;

  ListNode *rest = reverse(front->next); // reverse the whole list but the front
  front->next->next = front;             // update the next pointer of the 2nd element (now
                                         // the second to last element) to point to front
  front->next = NULL;                  // update the next pointer of front (now the
                                         // end of the list) to be NULL

  return rest;
}
```

11. Merge

Provided are two solutions for merging lists. The first one is recursive, and the second one is iterative.

```
ListNode *mergeLists(ListNode *one, ListNode *two) {
  if (one == NULL) return two;
  if (two == NULL) return one;

  // got this far? then neither list is empty
  if (one->data <= two->data) {
    one->next = mergeLists(one->next, two);
    return one;
  } else {
    two->next = mergeLists(one, two->next);
    return two;
  }
}
```

```
ListNode *mergeLists(ListNode *one, ListNode *two) {
  ListNode *merge = NULL;
  ListNode **mergePtr = &merge;

  while (one != NULL && two != NULL) {
    if (one->data <= two->data) {
      *mergePtr = one;
      one = one->next;
    } else {
      *mergePtr = two;
      two = two->next;
    }
    mergePtr = &((*mergePtr)->next);
  }

  if (one != NULL) *mergePtr = one;
  else *mergePtr = two;
  return merge;
}
```

Why might you preference one solution over the other? The recursive solution is a bit shorter, and depending on how comfortable you are with recursion versus advanced pointer techniques like double pointers, it might be easier to understand. However, the structure of this problem means you'll make a recursive call for every element in the finished list, which means you could end up with too many stack frames. Oftentimes, when you're doing "tail recursion" (when you make a single recursive call, at the very end of the function), it might actually be better to write an iterative solution, even if it's more complicated.

Note that some programming environments are able to optimize for tail recursion, meaning it's less of a performance hit to use it.

12. Draw Polygonal Path

```
void drawPolygonalPath(GWindow &window, PointNode *head) {
  if (head == NULL) {
    return;
  }

  PointNode *current = head;
  window.fillOval(current->x - 1, current->y - 1, 2, 2);

  while (current->next != NULL) {
    window.drawLine(current->x, current->y, current->next->x, current->next->y);
    window.fillOval(current->next->x - 1, current->next->y - 1, 2, 2);
    current = current->next;
    if (current == head) {
      break;
    }
  }
}
```

13. Braiding Lists

Provided are two solutions for braiding lists. The first one is iterative, and the second one is recursive.

```
void braid(ListNode *list) {
  ListNode *reverse = NULL;
  for (ListNode *curr = list;
            curr != NULL; curr = curr->next) {
    ListNode *newNode = new ListNode;
    newNode->data = curr->data;
    newNode->next = reverse;
    reverse = newNode;
  }

  // reverse now addresses a memory-independent
  //version of the original list,
  // where all of the nodes are in reverse order.
  ListNode *rest = reverse; //rest addresses part
                            //that has yet to be braided
  for (ListNode *curr = list;
            curr != NULL; curr = curr->next->next) {
    ListNode *next = rest->next;
    rest->next = curr->next;
    curr->next = rest;
    rest = next;
  }
}
```

```
void braid(ListNode *list, Queue<int> &numbers) {
  if (list == NULL) return;
  numbers.enqueue(list->value);
  braid(list->next, numbers);
  ListNode *newNode = new ListNode;
  newNode->data = numbers.dequeue();
  newNode->next = list->next;
  list->next = newNode;
}

void braid(ListNode *list) {
  Queue<int> numbers;
  braid(list, numbers);
}
```

Some of the same considerations apply when choosing between these two solutions as with merging lists. Note that this isn't actually tail recursion, however, it's still "unary recursion" (a single recursive call). Oftentimes problems solved using unary recursion have similarly complex iterative solutions.