# Jessica Lavin 2495543L

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Illustration for inserting a node to end of singly linked list (SLL):

# 

# Algorithm to insert node to end of singly linked list (SLL):

1. Initialise new node *n*, set *n*.next to null
2. If the SLL is empty:
   1. Set first to *n*
3. Else, if elements are present in the SLL:
   1. Initialise new node *current*, set *current* to first
   2. Traverse *current* till it is the last node in the SLL
   3. Set *curren*t.next to *n*
4. Return

The time complexity of the above algorithm is O(n) where n is the number of nodes present in the SLL.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Illustration for deleting alternate nodes in a singly linked list (SLL):

# 

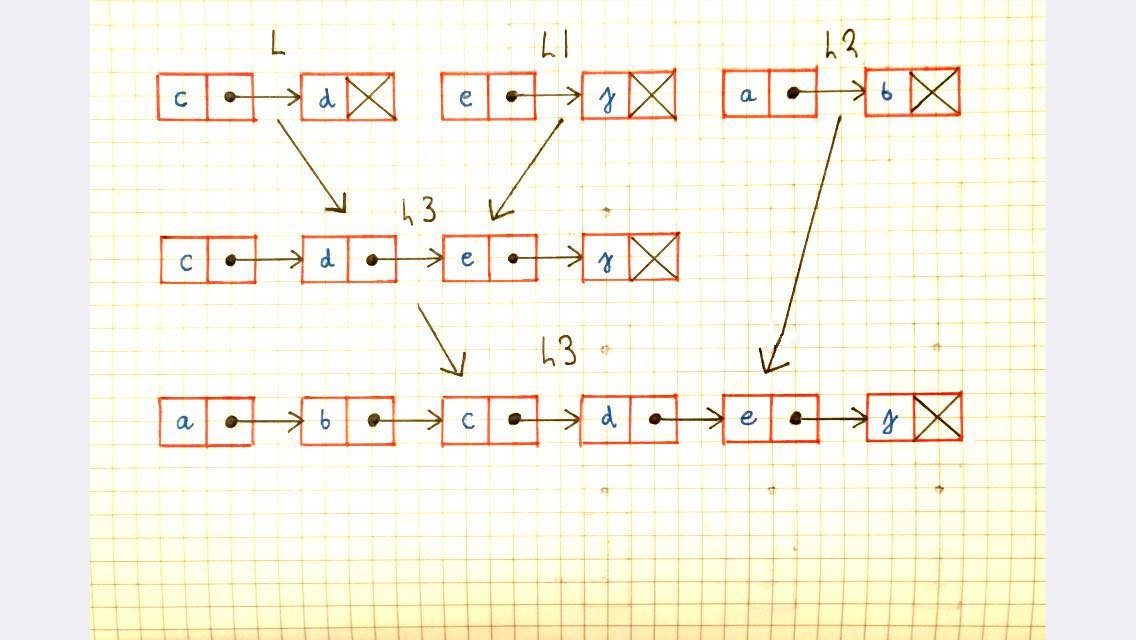
# Algorithm to delete alternate nodes in a singly linked list (SLL):

1. If the SLL is empty, return
2. Else, initialise new nodes *previous* and *current*, set *previou*s to first and *curren*t to first.next
3. While neither *previous* or *current* are empty, repeat:
   1. Free *current* by setting *previous*.next to *current*.next
   2. Empty the contents of *current*
   3. Set *previous* to *previous*.next
   4. If *previous* is not empty set *current* to *previous*.next
4. Return

The time complexity of the above algorithm is O(n) where n is the number of nodes present in the SLL.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Illustration for merging three sorted singly linked lists (SLL):



# Algorithm to merge three sorted singly linked lists (SLL):

1. Create new empty SLL, *L3*
2. Set *L3* to result of helper function mergeTwo (*L, L1*)
   1. Create new empty SLL, *temp*
   2. Initialise first node of list *a* and list *b*
   3. While *a* and *b* are not empty, repeat:
      1. If *a* < *b* add *a* to end of *temp,* set *a* to *a*.next
      2. Else add *b* to end of *temp*, set *b* to *b*.next
   4. While *a* is not empty, repeat:
      1. Add *a* to end of *temp*, set a to *a*.next
   5. While *b* is not empty, repeat:
      1. Add *b* to end of *temp*, set a to *b*.next
   6. Return *temp*
3. Set *L3* to result of helper function mergeTwo (*L2, L3*)
   1. Create new empty SLL, *temp*
   2. Initialise first node of list *a* and list *b*
   3. While *a* and *b* are not empty, repeat:
      1. If *a* < *b* add *a* to end of *temp,* set *a* to *a*.next
      2. Else add *b* to end of *temp*, set *b* to *b*.next
   4. While *a* is not empty, repeat:
      1. Add *a* to end of *temp*, set a to *a*.next
   5. While *b* is not empty, repeat:
      1. Add *b* to end of *temp*, set a to *b*.next
   6. Return *temp*
4. Remove any duplicates from *L3* using helper function (removeDups):
   1. Initialise new node *current* as *list*.first
   2. While current is not empty:
      1. Initialise new node *counter* as *current*
      2. While counter is not empty and *counter* equals *current*, set *counter* to *counter.*next
   3. Set *curren*t.next to *counter*
   4. Set *current* to *curren*t.next
   5. Return
5. Return *L3*

The time complexity of the above algorithm is O(n2) where n is the number of nodes in all 3 lists. This is because the insert tail method is called n times in this algorithm..

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_