Single Linked List End Node Deletion Using One Pointer

Auburn University - Software Construction
Austin Chase Minor

March 2, 2016

```
1:  function DELETE_LAST_NODE(Node* list)
2:      if list == null then
3:          return 0
4:      if list->next == null then
5:          delete list
6:          list = null
7:          return 0
8:      while list->next->next != null do
9:          list = list->next
10:     delete list->next
11:     list->next = null
12:     return 0
```

Proof

Loop Invarient

At the start of the while loop on line 8, list.next is guarenteed to exist.

Initialization

If the loop is reached in line 8, we know that (i) list is not null and (ii) list.next is not null. (i) follows from lines 2-3 and (ii) follows from lines 4 - 7. Thus for (i) nothing is deleted. Thus for (ii) the last node (which happens to be the first node) is deleted. Thus the loop invarient holds before the first time the loop is executed.

Maintenance

The loop has to possibilities. Either (i) list.next.next does not exist (null) or (ii) list.next.next does exist. In (i), the loop breaks and we know from the loop invarient that list.next exists (since no work was done). In (ii), list.next.next does exist. Thus line 9 executes setting list to list.next. Since list.next.next exists and list = list.next we know that the new list.next must exist. Thus the loop holds for maintenace.

Termination

Since the loop invarient holds for maintenance, we know that at the end of the loop that (i) list.next exists and (ii) list.next.next does not exist. Thus, list.next is the last element. This element is thus deleted and set to null. Furthermore, it can be easily shown that this strategy applies to circular lists. Just replace line 11 with list.next = head_node and replace the check in line 8 with list.next.next != head_node. Therefore, this algorithm is correct.

Efficiency

For comparison purposes presented bellow is the algorithm for a different two node version of delete_last_node modified from Dr. Qin C++ lecture slides.

```
1:  function DELETE_LAST_NODE(Node* list)
2:      Node* (cur_ptr, pre_ptr) = null
3:      if list == null then
4:          return 0
5:      if list.next == null then
6:          delete list
7:          list = null
8:          return 0
9:      pre_ptr = list;
10:     cur_ptr = list->next;
11:     while cur_ptr->next != null do
12:         pre_ptr = cur_ptr;
13:         cur_ptr = cur_ptr->next;
14:     delete list->next
15:     list->next = null
16:     return 0
```

Obviously some simplications were made in the algorithms listed above compared to real implementations (dealing with pointer references, etc.). However, the proof and general memory/speed analysis is well approximated.

Memory Efficiency

Through analysis it is obvious that the single node algorithm presented needs only space for one pointer. In c++, it would need space for two pointers because of the pointer reference issue. The double node algorithm in c++ would require the use of three pointers: list, cur_ptr, and pre_ptr. Thus even in c++, the single node algorithm is more space efficient by one address variable.

Speed Efficiency

Establishing constants on every needed line of the single node algorithm

and assuming for simplicity that every if statement executes yields: $T(N) = c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + (K_N + 1) * c_8 + K_N * c_9 + c_{10} + c_{11} + c_{12}$ Where K_N is the number of items in the list less 2. I.E. the algorithm stops on the next, next to last element. Applying some napkin calculations to the time constants yields: $T(N) = 2+1+4+5+2+1+1+(N-1)*5+(N-2)*3+3+3+1 = 8N+12$.

Doing a similar analysis for the two node algorithm yields: $T(N) = 2+2+1+4+2+1+1+2+4+(N-1)*4+(N-2)*2+(N-2)*4+3+3+1 = 10N+8$

Thus using back of the napkin speed efficiency analysis the one node ($T(N) = 8N + 12$) and two node ($T(N) = 10N + 8$) algorithms have similar speed efficiency. Thus the one node algorithm performs similarly to the two node algorithm with a added space efficiency.