ECE 264 Advanced C Programming

2009/02/20

Contents

1 Search and Delete in Linked List

The linked list is interesting but not particularly useful if we can only insert, not search nor delete. The next example shows how to search and delete. Each returns 1 if the value is found in the current list, and 0 if the value is not in the current list.

```
#ifndef LISTNODE2_H
#define LISTNODE2_H
#include "listnode.h"
int List_search(Node * list, int v, Node * * n);
int List_delete(Node * * list, int v);
#endif
```

Notice that we **extend** the previous header file by declaring two additional functions in **another** header file (called `listnode2.h`). C allows us to add more functions without changing the existing header files. This is very convenient because we do not have to rewrite or copy existing header files. This new header file includes the previous one so that the structure is known in the new header file.

```
#include "listnode.h"
#include "listnode2.h"
#include <stdio.h>
#include <stdlib.h>
int List_search(Node * list, int v, Node * * n)
{
  /* is v inside the list? If not, return 0. */
  /* If so, return 1 and n points to the node. */
  /* Node n points to part of list, not a separate list.
Therefore,
     we should not intend to delete the list starting at n */
  Node * curr = list;
```

```
  (* n) = 0;
  while ((curr != 0) && ((curr -> ln_value != v)))
    /* must check curr first */
    {
      curr = curr -> ln_next;
    }
  if (curr == 0) /* cannot find it */
    {
      return 0;
    }
  (* n) = curr;
  return 1;
}

int List_delete(Node * * list, int v)
{
  /* return 1 if found and delete */
  /* return 0 if not found */
  Node * curr = (* list);
  Node * prev = (* list);
  while ((curr != 0) && ((curr -> ln_value != v)))
    {
      prev = curr;
      curr = curr -> ln_next;
    }
  if (curr == 0) /* not found */
    {
      return 0;
    }
  if (curr == (* list)) /* first node */
    {
      (* list) = (* list) -> ln_next;
    }
  else
    {
      prev -> ln_next = curr -> ln_next;
    }
  Node_destruct(curr);
  return 1;
}
```

`List_search` takes three arguments: the first is the pointer of the list to be searched, the second is the value to be searched, and the third is the pointer of the node if the value is

found. If the value is not in the list, the third argument is assigned to zero. The function uses another pointer called `curr` to represent the current location at the list. If the pointer is not zero, we check whether the value is the same as the second input argument. If they are different, move to the next node. The `while` block terminates in one of the two conditions: `curr` is zero or the value is found. In the former case, the value is not found and the function returns zero. In the latter case, we assign `* n` to the location of the current node and return 1. This `while` block is another example of using "short-circuit evaluation" in C. We must check whether `curr` is zero first. If it is zero, the second condition is **not** checked. We **cannot** exchange the order of the two conditions. Doing so will cause the program to crash when the searched value is not in the list.

A linked list is called a *dynamic structure* because its size grows and shrinks as needed when the program is running. `List_delete` is a little more complicated because we have to remember the node to be deleted and the node that is pointing to the deleted node. A figure on page 459 of ABoC illustrates the procedure of deletion. This function maintains two pointers, one called `curr` and the other called `prev`. The former checks whether a node has the value we want to delete. If it does not, we store the current node in `prev` and proceed to the next node. The `while` block terminates because either we reach the end of the list, or we have found the value. If we reach the end, the value is not found and the function returns zero. Otherwise, we check whether the node to be deleted is the very first one. If so, move the pointer to its next. If it is not the first one, link the previous node's next to the current node's next. Remember to release the memory held by the current node. Since the location pointed by `list` may change, we **have to** pass the address of `list` to ensure that it is correctly updated.

The `main` function tests `List_search` and `List_delete`. Two things to notice here: (1) `list2` points to part of `list1` if a value is found. Therefore, we call `List_destruct(list1)` and do **not** call `List_destruct(list2)`. Doing so would delete the same list twice. (2) We should test searching and deleting the first and the last values in the list. This is often necessary because we treat them differently from the rest of the nodes. In Exercise 7, you will be asked to use test coverage to see whether your program tests every condition.

```
#include "listnode.h"
#include "listnode2.h"
#include <stdio.h>
void testFunc3()
{
  Node * list1 = 0;
  Node * list2 = 0;
  int cnt;
  for (cnt = 0; cnt < 10; cnt ++)
    {
      list1 = List_insert(list1, cnt + 10);
```

```
    }
  List_print(list1);
  printf("search %d = %d\n", 6, List_search(list1, 6, & list2));
  List_print(list2);
  printf("search %d = %d\n", 16, List_search(list1, 16, & list2));
  List_print(list2);
  printf("search %d = %d\n", 19, List_search(list1, 19, & list2));
  List_print(list2);
  printf("search %d = %d\n", 10, List_search(list1, 10, & list2));
  List_print(list2);

  printf("delete %d = %d\n", 3, List_delete(& list1, 3));
  List_print(list1);
  printf("delete %d = %d\n", 13, List_delete(& list1, 13));
  List_print(list1);

  printf("delete %d = %d\n", 10, List_delete(& list1, 10));
  List_print(list1);

  printf("delete %d = %d\n", 19, List_delete(& list1, 19));
  List_print(list1);

  List_destruct(list1);
  /* do not List_destruct(list2) because it points to one node
     in list1 */
}

int main(int argc, char * argv[])
{
  testFunc3();
  return 0;
}

/*
  19 18 17 16 15 14 13 12 11 10

  search 6 = 0


  search 16 = 1
  16 15 14 13 12 11 10
```

```
  search 19 = 1
  19 18 17 16 15 14 13 12 11 10

  search 10 = 1
  10

  delete 3 = 0
  19 18 17 16 15 14 13 12 11 10

  delete 13 = 1
  19 18 17 16 15 14 12 11 10

  delete 10 = 1
  19 18 17 16 15 14 12 11

  delete 19 = 1
  18 17 16 15 14 12 11

*/

# Makefile
# If you do not understand Makefile, review Exercise 3.
# This is a comment (after #).
OBJS = listnode.o listnode2.o list2main.o
SRCS = listnode.c listnode2.c list2main.c
CFLAGS = -g -Wall
GCC = gcc $(CFLAGS)
TARGET = list2main
list2main: $(OBJS)
           $(GCC) $(OBJS) -o $(TARGET) # create executable
           ./$(TARGET) # execute the program
           valgrind --leak-check=yes ./$(TARGET) # memory leak?
.c.o:
        $(GCC) -c $*.c
clean:
        rm -f $(OBJS) $(TARGET)
depend:
        makedepend $(SRCS)
```