ECE 264 Advanced C Programming

Contents

1 Linked List for String

We have studied linked list for integers:

```
typedef struct listnode
{
  struct listnode * ln_next;
  int ln_value;
} Node;
```

We can modify the node so that each stores a string:

```
typedef struct listnode
{
  struct listnode * ln_next;
  char * ln_string;
} Node;
```

```
#ifndef LISTNODE_H
#define LISTNODE_H
typedef struct listnode
{
  struct listnode * ln_next;
  char * ln_string;
} Node;

Node * Node_construct(char * str);
```

```
Node * List_copy(Node * n);
void List_assign(Node * * n1, Node * n2);
void Node_destruct(Node * n);
void List_destruct(Node * n);
Node * List_insert(Node * n, char * str);
int List_search(Node * list, char * str, Node * * n);
int List_delete(Node * * list, char * str);
char * Node_getString(Node * n);
void Node_print(Node * n);
void List_print(Node * n);
#endif

#include "listnode.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
Node * Node_construct(char * str)
{
  Node * n = malloc(sizeof(Node));
  n -> ln_string = malloc((strlen(str) + 1) * sizeof(char));
  strcpy(n -> ln_string, str);
  n -> ln_next = 0;
  return n;
}

Node * List_insert(Node * n, char * str)
{
  Node * p = Node_construct(str);
  p -> ln_next = n;
  return p;
}

Node * List_copy(Node * n2)
{
  Node * next = n2;
  Node * n;
  if (n2 == 0) { return 0; }
  n = Node_construct(n2 -> ln_string);
  next = next -> ln_next;
  while (next != 0)
    {
      n = List_insert(n, next -> ln_string);
      next = next -> ln_next;
```

```
    }
  return n;
}

void List_assign(Node * * n1, Node * n2)
{
  if ((* n1) == n2)
    { return; }
  List_destruct(* n1);
  * n1 = List_copy(n2);
}

void Node_destruct(Node * n)
{
  free (n -> ln_string);
  free (n);
}

void List_destruct(Node * n)
{
  Node * prev = n;
  Node * next;
  while (prev != 0)
    {
      next = prev -> ln_next;
      Node_destruct(prev);
      prev = next;
    }
}

char * Node_getString(Node * n)
{
  return (n -> ln_string);
}

void Node_print(Node * n)
{
  printf("%s ", n -> ln_string);
}

void List_print(Node * n)
{
```

```
  Node * curr = n;
  while (curr != 0)
    {
      Node_print(curr);
      curr = curr -> ln_next;
    }
  printf("\n\n");
}

int List_search(Node * list, char * str, Node * * n)
{
  Node * curr = list;
  (* n) = 0;
  while ((curr != 0) && ((strcmp(curr -> ln_string, str) != 0)))
    {
      curr = curr -> ln_next;
    }
  if (curr == 0)
    {
      return 0;
    }
  (* n) = curr;
  return 1;
}

int List_delete(Node * * list, char * str)
{
  Node * curr = (* list);
  Node * prev = (* list);
  while ((curr != 0) && ((strcmp(curr -> ln_string, str) != 0)))
    {
      prev = curr;
      curr = curr -> ln_next;
    }
  if (curr == 0)
    {
      return 0;
    }
  if (curr == (* list))
    {
      (* list) = (* list) -> ln_next;
    }
```

```
  else
    {
      prev -> ln_next = curr -> ln_next;
    }
  Node_destruct(curr);
  return 1;
}

#include "listnode.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char * argv[])
{
  Node * list1 = 0;
  Node * list2 = 0;
  int numString = 20;
  char * * words;
  int cnt;
  words = malloc(numString * sizeof(char* ));
  for (cnt = 0; cnt < numString / 2; cnt ++)
    {
      words[cnt] = malloc(80 * sizeof(char));
      sprintf(words[cnt], "word%d", cnt);
      list1 = List_insert(list1, words[cnt]);
    }
  for (cnt = numString / 2; cnt < numString; cnt ++)
    {
      words[cnt] = malloc(80 * sizeof(char));
      sprintf(words[cnt], "word%d", cnt);
    }
  List_print(list1);
  printf("search \"%s\" = %d\n", words[6],
         List_search(list1, words[6], & list2));
  List_print(list2);
  printf("search \"%s\" = %d\n", words[14],
         List_search(list1, words[14], & list2));
  List_print(list2);
  printf("search \"%s\" = %d\n", words[3],
         List_search(list1, words[3], & list2));
  List_print(list2);
  printf("search \"%s\" = %d\n", words[12],
         List_search(list1, words[12], & list2));
```

```
  List_print(list2);

  printf("delete \"%s\" = %d\n", words[4],
         List_delete(& list1, words[4]));
  List_print(list1);
  printf("delete \"%s\" = %d\n", words[13],
         List_delete(& list1, words[13]));
  List_print(list1);

  printf("delete \"%s\" = %d\n", words[2],
         List_delete(& list1, words[2]));
  List_print(list1);

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

  List_destruct(list1);

  for (cnt = 0; cnt < numString; cnt ++)
    {
      free(words[cnt]);
    }
  free(words);
  return 0;
}

/*
  output:

  word9 word8 word7 word6 word5 word4 word3 word2 word1 word0

  search "word6" = 1
  word6 word5 word4 word3 word2 word1 word0

  search "word14" = 0


  search "word3" = 1
  word3 word2 word1 word0

  search "word12" = 0
```

```
  delete "word4" = 1
  word9 word8 word7 word6 word5 word3 word2 word1 word0

  delete "word13" = 0
  word9 word8 word7 word6 word5 word3 word2 word1 word0

  delete "word2" = 1
  word9 word8 word7 word6 word5 word3 word1 word0

  delete "word19" = 0
  word9 word8 word7 word6 word5 word3 word1 word0

 */
```

Most of the code is the same for the code to handle integer values, except the places where the values are handled (replaced by `ln_string`). The constructor needs to allocate memory for the string and copy the content:

```
  Node * n = malloc(sizeof(Node));
  n -> ln_string = malloc((strlen(str) + 1) * sizeof(char));
  strcpy(n -> ln_string, str);
```

The destructor has to release the memory for the string before releasing the memory for the node:

```
  free (n -> ln_string);
  free (n);
```

Please notice that the constructor and the destructor are **symmetric**. In `List_search`, `strcmp` is used to determine whether a node holds the string:

```
   while ((curr != 0) && ((strcmp(curr -> ln_string, str) != 0)))
```

2 Linked List for Structure

We can further extend the linked list so that each node holds the data for a structure.

```
/* person.h */
#ifndef PERSON_H
#define PERSON_H
typedef struct
{
  int p_age;
  char * p_name;
} Person;

Person * Person_construct(int a, char * n); /* return pointer */
Person * Person_copy(Person * p);
void Person_assign(Person * * p1, Person * p2);
/* notice * * for the first object */
void Person_destruct(Person * p);
void Person_print(Person * p);
int Person_getAge(Person * p);
char * Person_getName(Person * p);
#endif
```

Each node in the linked list has a pointer for a `Person` object.

```
typedef struct listnode
{
  struct listnode * ln_next;
  Person * ln_person;
} Node;
```

We can modify the constructor of each node:

```
Node * Node_construct(int a, char * name)
{
     Node * n = malloc(sizeof(Node);
     Person * p = Person_construct(a, name);
     n -> ln_person = p;
     n -> ln_next = 0;
     return n;
}
```

The destructor is symmetric:

```
void Node_destruct(Node * n)
{
     Person_destruct(n -> ln_person);
     free (n);
}
```

As you can see, we are **reusing** the functions for `Person` so that we do not have to rewrite them. Reusing code is one of the most important techniques to improve productivity and to prevent mistakes. This example also shows a **hierarchy** to write code. The linked list is built upon the `Person` structure. When you develop a large program, you usually break the program into smaller units, write each unit, and then integrate them.