ECE 264 Advanced C Programming

Contents

1 Stack

Our current implement of linked list puts the newly added value at the beginning of the list. As a result, it is "first-in-last-out" (FILO). This is called a *stack*. Why do we care about stacks? Because it is used in *every* program. You have been using it, even though you may not be aware of it. Consider the following example of function calls.

```
func1()
{
   ...
   func2();
   ...
}
```

```
func2()
{
    ...
    func3();
    ...
}
```

```
func3()
{
   ...
   ...
}
```

When `func3` completes, where does the program continue execution? In `func2`, just after the place where `func3` was called earlier. When `func2` completes, where does the program continue execution? In `func1`, just after the place where `func2` was called earlier. As you can see, `func1` is called *before* `func2` is called, `func2` is called *before* `func3` is called. The program returns to `func2` after `func3` completes and returns to `func1` after `func2` completes. Imagine that we put the location just below each function call into a *stack*. Every time the program calls a function, the location is *pushed* into the stack. Every time a function completes, the *top* of the stack is *popped* to determine where to continue. We can implement a stack using a linked list. A stack allows us to insert or delete **only** at the top of the stack. They are called *push* and *pop* respectively. We are not allowed to modify anything below the top.

```
/* stacknode.h */
#ifndef STACKNODE_H
#define STACKNODE_H
typedef struct stacknode
{
  struct stacknode * s_next;
  int s_value;
} Node;

void Stack_push(Node * * n, int v);
int Stack_top(const Node * n);
int Stack_pop(Node * * n);
void Stack_destruct(Node * n);
#endif

/* stacknode.c */
#include "stacknode.h"
#include <stdio.h>
#include <stdlib.h>
/* static: This function is visible to this file only */
static Node * Stack_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> s_value = v;
  n -> s_next = 0;
  return n;
}

void Stack_push(Node * * n, int v)
{
  Node * p = Stack_construct(v);
  p -> s_next = (* n);
  (* n) = p;
}

int Stack_pop(Node * * n)
{
  Node * top;
  int val;
  if ((* n) == 0)
    {
      printf("stack underflow\n");
      return 0;
```

```
    }
  top = (* n);
  val = top -> s_value;
  (* n) = (* n) -> s_next;
  free (top);
  return val;
}


static void Node_destruct(Node * n)
{
  free (n);
}

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

int Stack_top(const Node * n)
{
  if (n == 0)
    {
      printf("stack underflow\n");
      return 0;
    }
  return (n -> s_value);
}

#include "stacknode.h"
#include <stdio.h>
void testFunc()
{
  Node * stack1 = 0;
  /*
    Node * stack2 = Stack_construct(10);
```

```
    undefined reference to 'Stack_construct'
    because it is static, invisible to other files
  */
  int cnt;
  for (cnt = 0; cnt < 10; cnt ++)
    {
      Stack_push(& stack1, cnt + 100);
    }

  printf("top = %d\n", Stack_top(stack1));
  printf("pop = %d\n", Stack_pop(& stack1));
  printf("pop = %d\n", Stack_pop(& stack1));
  printf("pop = %d\n", Stack_pop(& stack1));
  Stack_push(& stack1, 200);
  Stack_push(& stack1, 300);
  printf("pop = %d\n", Stack_pop(& stack1));
  printf("pop = %d\n", Stack_pop(& stack1));
  printf("pop = %d\n", Stack_pop(& stack1));
  printf("pop = %d\n", Stack_pop(& stack1));
  Stack_destruct(stack1);
}

/*
  top = 109
  pop = 109
  pop = 108
  pop = 107
  pop = 300
  pop = 200
  pop = 106
  pop = 105
*/
int main(int argc, char * argv[])
{
  testFunc();
  return 0;
}
```

Implementing a stack is very similar to implementing a linked list. We have removed the search and delete functions because a stack allows us to check the top only. We are not allowed to see anything other than the top. For simplicity, I do not include the copy and the assign functions. You should be able to add them without any problem. A stack supports only three functions: `push`, `top`, and `pop`. We need some functions to help.

Therefore, we declare and create the functions inside `stacknode.c` and declare them as `static`. If a function is `static`, **it is visible to only the file and not outside**. Static functions reduce the chance of *name collision*: two functions in two files have the same name. We will not be allowed to use the static functions outside the file. We have a smaller chance to make mistakes and can spend less time finding and correcting mistakes. A stack can be popped as many times as pushed. If we try to pop more, we encounter *stack underflow* and the program reports that.

2 Queue

A stack is "First-In-Last-Out". A stack is useful in handling function call returns. However, it is not so great when you want to buy movie tickets. It certainly isn't very nice if the last person gets the tickets first. What we need is "First-In-First-Out" (FIFO). It is called a *queue*. A queue also has three operations:

- *enqueue*: insert an element at the end.
- *dequeue*: remove the element from the beginning.
- *front*: retrieve the element at the beginning without removing it.

```
/* queuenode.h */
#ifndef QUEUENODE_H
#define QUEUENODE_H
typedef struct queuenode
{
  struct queuenode * q_next;
  int q_value;
} Node;

void Queue_enqueue(Node * * n, int v);
int Queue_front(const Node * n);
int Queue_dequeue(Node * * n);
void Queue_destruct(Node * n);
#endif

/* queuenode.c */
#include "queuenode.h"
#include <stdio.h>
#include <stdlib.h>
static Node * Queue_construct(int v)
```

```
{
  Node * n = malloc(sizeof(Node));
  n -> q_value = v;
  n -> q_next = 0;
  return n;
}

void Queue_enqueue(Node * * n, int v)
{
  Node * curr = (* n);
  Node * p = Queue_construct(v);
  if (curr == 0) /* currently empty queue */
    {
      (* n) = p;
      return;
    }
  /* curr is not zero */
  while (curr -> q_next != 0)
    {
      curr = curr -> q_next;
    }
  curr -> q_next = p;
}

int Queue_dequeue(Node * * n)
{
  Node * front;
  int val;
  if ((* n) == 0)
    {
      printf("queue underflow\n");
      return 0;
    }
  front = (* n);
  val = front -> q_value;
  (* n) = (* n) -> q_next;
  free (front);
  return val;
}

static void Node_destruct(Node * n)
{
```

```
  free (n);
}

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

int Queue_front(const Node * n)
{
  if (n == 0)
    {
      printf("queue underflow\n");
      return 0;
    }
  return (n -> q_value);
}

#include "queuenode.h"
#include <stdio.h>
void testFunc()
{
  Node * queue1 = 0;
  int cnt;
  for (cnt = 0; cnt < 10; cnt ++)
    {
      Queue_enqueue(& queue1, cnt + 100);
    }
  printf("front = %d\n", Queue_front(queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  Queue_enqueue(& queue1, 200);
  Queue_enqueue(& queue1, 300);
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
```

```
  printf("front = %d\n", Queue_front(queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("front = %d\n", Queue_front(queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  printf("dequeue = %d\n", Queue_dequeue(& queue1));
  Queue_destruct(queue1);
}

/*
  front = 100
  dequeue = 100
  dequeue = 101
  dequeue = 102
  dequeue = 103
  dequeue = 104
  front = 105
  dequeue = 105
  dequeue = 106
  front = 107
  dequeue = 107
  dequeue = 108
  dequeue = 109
  dequeue = 200
*/
int main(int argc, char * argv[])
{
  testFunc();
  return 0;
}
```

`Queue_dequeue` is actually very similar to `Stack_pop` because both functions remove the first element. The main difference between a stack and a queue is `Stack_push` and `Queue_enqueue`. Push needs to modify the very first node but enqueue has to **traverse the whole queue** to find the end and add the new node at the end. That is the reason of the `while` block. It finds the node whose next is zero. This node must be the last node in the queue. Then, the newly created node is added as the next node. It is not very efficient if we have to traverse the whole list for adding a value. One solution is to keep another pointer to the end of the list.

Exercise: Write the code for `Queue` that maintains two pointers for the head and the tail so that insertion does not traverse the whole list.

3 Sorting using Linked List

We can use a linked list to sort numbers.

```
/* listnode.h */
#ifndef LISTNODE_H
#define LISTNODE_H
typedef struct listnode
{
  struct listnode * ln_next;
  int ln_value;
} Node;

Node * List_copy(Node * n);
void List_assign(Node * * n1, Node * n2);
void List_insert(Node * * n, int v);
int List_delete(Node * * list, int v);
void List_print(Node * n);
void List_destruct(Node * n);
int List_search(Node * list, int v);
#endif
```

```
/* listnode.c */
#include "listnode.h"
#include <stdio.h>
#include <stdlib.h>
static Node * Node_construct(int v)
{
  Node * n = malloc(sizeof(Node));
  n -> ln_value = v;
  n -> ln_next = 0;
  return n;
}

void List_insert(Node * * n, int v)
{
  Node * p = Node_construct(v);
  Node * curr = * n;
```

```
  Node * prev = * n;
  if (( *n) == 0)
    {
      *n = p;
      return;
    }
  while ((curr != 0) && ((curr -> ln_value) < v))
    {
      prev = curr;
      curr = curr -> ln_next;
    }
  if (curr == (* n)) /* first */
    {
      p -> ln_next = (* n);
      * n = p;
    }
  else
    {
      p -> ln_next = prev -> ln_next;
      prev -> ln_next = p;
    }
}

Node * List_copy(Node * n2)
{
  Node * next = n2;
  Node * n = 0;
  while (next != 0)
    {
      List_insert(& n, next -> ln_value);
      next = next -> ln_next;
    }
  return n;
}

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

```
static void Node_destruct(Node * n)
{
  free (n);
}

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

static void Node_print(Node * n)
{
  printf("%d ", n -> ln_value);
}

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, int v)
{
  Node * curr = list;
  while ((curr != 0) && ((curr -> ln_value) < v))
    /* must check curr first */
    {
      curr = curr -> ln_next;
    }
```

```
  if ((curr != 0) && ((curr -> ln_value) == v))
    {
      return 1;
    }
  return 0;
}

int List_delete(Node * * list, int v)
{
  Node * curr = (* list);
  Node * prev = (* list);
  while ((curr != 0) && ((curr -> ln_value) < v))
    {
      prev = curr;
      curr = curr -> ln_next;
    }
  if ((curr != 0) && ((curr -> ln_value) == v))
    {
      if (curr == (* list)) /* first node */
        {
          (* list) = (* list) -> ln_next;
        }
      else
        {
          prev -> ln_next = curr -> ln_next;
        }
      free (curr);
      return 1;
    }
  return 0;
}

/* listmain.c */
#include "listnode.h"
#include <stdio.h>
void testFunc()
{
  Node * list1 = 0;
  Node * list2 = 0;
  int data [] = {1, 100, 3, 64, 5, -6, 999, 4, -85, 7};
  int numElem = sizeof(data) / sizeof(int);
  int cnt;
  for (cnt = 0; cnt < numElem; cnt ++)
```

```
  {
    printf("inserting %d\n", data[cnt]);
    List_insert(& list1, data[cnt]);
    List_print(list1);
  }
list2 = List_copy(list1);
List_print(list2);
printf("search %d = %d\n", 6, List_search(list1, 6));
List_print(list1);
printf("search %d = %d\n", 5, List_search(list1, 5));
List_print(list1);

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);

printf("delete %d = %d\n", -85, List_delete(& list1, -85));
List_print(list1);

printf("delete %d = %d\n", 999, List_delete(& list1, 999));
List_print(list1);

printf("delete %d = %d\n", -6, List_delete(& list1, -6));
List_print(list1);

printf("delete %d = %d\n", 100, List_delete(& list1, 100));
List_print(list1);

for (cnt = 0; cnt < numElem; cnt ++)
  {
    printf("inserting %d\n", data[cnt] * cnt);
    List_insert(& list1, data[cnt] * cnt);
    List_print(list1);
  }
```

```
  printf("search %d = %d\n", 6, List_search(list1, 6));
  List_destruct(list1);
  List_destruct(list2);
}

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

/*

  inserting 1
  1

  inserting 100
  1 100

  inserting 3
  1 3 100

  inserting 64
  1 3 64 100

  inserting 5
  1 3 5 64 100

  inserting -6
  -6 1 3 5 64 100

  inserting 999
  -6 1 3 5 64 100 999

  inserting 4
  -6 1 3 4 5 64 100 999

  inserting -85
  -85 -6 1 3 4 5 64 100 999

  inserting 7
  -85 -6 1 3 4 5 7 64 100 999
```

```
-85 -6 1 3 4 5 7 64 100 999

search 6 = 0
-85 -6 1 3 4 5 7 64 100 999

search 5 = 1
-85 -6 1 3 4 5 7 64 100 999

delete 3 = 1
-85 -6 1 4 5 7 64 100 999

delete 13 = 0
-85 -6 1 4 5 7 64 100 999

delete 10 = 0
-85 -6 1 4 5 7 64 100 999

delete 19 = 0
-85 -6 1 4 5 7 64 100 999

delete -85 = 1
-6 1 4 5 7 64 100 999

delete 999 = 1
-6 1 4 5 7 64 100

delete -6 = 1
1 4 5 7 64 100

delete 100 = 1
1 4 5 7 64

inserting 0
0 1 4 5 7 64

inserting 100
0 1 4 5 7 64 100

inserting 6
0 1 4 5 6 7 64 100
```

```
  inserting 192
  0 1 4 5 6 7 64 100 192

  inserting 20
  0 1 4 5 6 7 20 64 100 192

  inserting -30
  -30 0 1 4 5 6 7 20 64 100 192

  inserting 5994
  -30 0 1 4 5 6 7 20 64 100 192 5994

  inserting 28
  -30 0 1 4 5 6 7 20 28 64 100 192 5994

  inserting -680
  -680 -30 0 1 4 5 6 7 20 28 64 100 192 5994

  inserting 63
  -680 -30 0 1 4 5 6 7 20 28 63 64 100 192 5994

  search 6 = 1

*/
```

The main difference is comparing a node's value with the inserted value. We will move across the nodes until reaching the first node whose value is larger than the inserted value.

```
  while ((curr != 0) && ((curr -> ln_value) < v))
```

Similarly, when we search or delete, we need to check until reaching a value that is larger than the input value.