Fundamentals of Programming 2

Circular Doubly Linked List

Arkadiusz Chrobot

Department of Computer Science

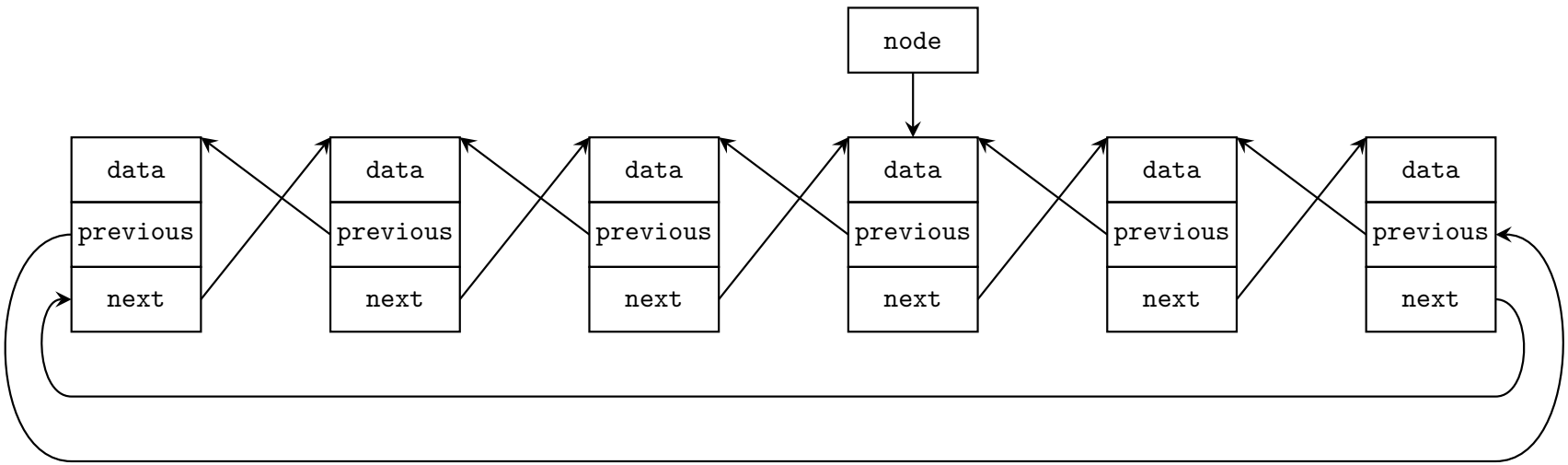
May 14, 2019

Outline

Introduction

A circular linked list is a list that doesn't have the first and the last element. Every node has its predecessor and successor. Such a list can be singly or doubly linked. A singly linked linear list can be converted to a circular list by storing the address of the first element of the list in the pointer field of the last element. Similar operation allow for converting the doubly linked linear list into the circular list. In the lecture a program is presented that uses a list that from the beginning is a circularly linked list. The next slide contains a figure that shows the schema of such a list.

Introduction



The Circular Doubly Linked List

Implementation

Just like in the case of previously discussed lists, the circular doubly linked list is explained with the use of a program which uses it to store natural numbers in the ascending order. Since the circular list doesn't have the beginning or the end, the order is relative, i.e. depends on the element that currently stores the smallest number in the list.
The circular list can be implemented with the use of an array or in a form of dynamically allocated data structure. The latter is described in the lecture. Such lists are often implemented as lists with sentinels, for simplifying the operations carried out on them. The presented implementation doesn't have such features.

Base Type and List Pointer

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  const unsigned int how_many = 2;
5
6  struct list_node
7  {
8      int data;
9      struct list_node *previous, *next;
10 } *list_pointer;
```

Base Data Type

The base data type of circular doubly linked list is the same as for the doubly linked linear list. The type can also be modified to make it better suitable for a specific problem. The circular doubly linked list doesn't have any element that contains in its pointer fields even a single `NULL` value.
In the previous slide, aside the definition of the base data type and declaration of the list pointer, there are also preprocessor statements that include to the program the same header files as in the previous programs that used various kinds of lists. Also there is defined a constant that is passed as an argument for a function that displays the values stored in the list on screen.

List Pointer

In the case of a circular list, regardless of it is a singly or doubly linked list, the list pointer can point any element that belongs to the list. There are no requirements concerning which one it is. If the list pointer is an empty pointer (it has the `NULL` value), it means that the list is empty (nonexistent).

List Operations

The circular doubly linked list is an abstract data structure, just like the other already described lists. Aside of defining the base data type for the list also operations that are carried out on the list have to be implemented. Just like in the case of previously discussed lists in the lecture only the basic operations for the circular doubly linked list are described: creating the list, adding an element (together with traversing the list in search for an element), removing a single element, displaying the content of the list and removing the whole list.

Creating the List

```
1  struct list_node *create_list(int number)
2  {
3      struct list_node *new_node = (struct list_node *)
4                                   malloc(sizeof(struct list_node));
5      if(new_node) {
6          new_node->data = number;
7          new_node->previous = new_node->next = new_node;
8      }
9      return new_node;
10 }
```

Creating the List

The function presented in the previous slide creates a circular doubly linked list by creating its first element. The definition of the function is similar to the definition of its counterpart for the doubly linked linear list. The only difference is the initialization of the field pointers of the element. They are assigned the address of the element that contains them (7th line) instead of the `NULL` value. In that way the circular doubly linked list with a single element is created.

Adding an Element to the List

The operation of adding an element to the circular doubly linked list has to be carried out for a nonempty list. Just like in the cases of previously discussed lists, the circular doubly linked list ought to store values in the ascending order. The problem of finding a place in the list for a new element is reduced to finding an element in the list that stores a value greater than the value stored in the new element or to finding the element that stores the smallest value in the whole list. In the latter case the new element can store a grater value than any of the element of the list. Both operations of searching for the aforementioned elements are implemented in separated helper functions that are presented in the next slides.

Adding an Element of the List

Finding the Smallest Value in the List

```
1  struct list_node *find_minimum_value_node(
2                                      struct list_node *list_pointer)
3  {
4          struct list_node *start, *result;
5          start = result = list_pointer;
6          int minimum = list_pointer->data;
7          do  {
8                  if(minimum>list_pointer->data) {
9                      minimum = list_pointer->data;
10                     result = list_pointer;
11                 }
12                 list_pointer = list_pointer->next;
13         } while(list_pointer!=start);
14         return result;
15 }
```

Adding an Element to the List

Finding the Smallest Value in the List

The function presented in the previous slide searches for an element that stores the currently smallest number in the list. If such a value repeats in the list several times or in other words in the list is an aggregation or *cluster* of elements that store such a value then the function will find the first element of the cluster. The `find_minimum()` function uses a modified version of the algorithm of searching the smallest value in an unsorted array. The function takes the list pointer as an argument and returns the address of the element that stores the smallest value. The applied algorithms requires traversing all the elements of the list in a loop. Since none of the elements of the list stores a `NULL` value in any of the pointer fields, a condition that allows the loop to terminate has to be formulated differently than in cases of the previously discussed lists.

Adding an Element To the List

Finding the Smallest Value in the List

In the 4th line of the `find_minimum_value_node()` function are declared two local pointers: `start` and `result`. In the next line the current value of list pointer is assigned to both of them. The `start` pointer stores an address of the element from which the function starts traversing the list. The `list_pointer` variable is used for pointing successive elements of the list inside the loop. If value of the pointer becomes the same as the value of the `start` then it means that all the element of the list has been visited and the loop should be terminated. The loop should perform at least one iteration, so the described condition have to be evaluated after the loop's body, hence the `do...while` is applied in the function instead of the `while` loop. After the loop terminates the `result` pointer points to the element that stores the smallest number in the whole list. The address is returned by the function (line no. 14). The local variable `minimum` is used for storing the smallest number. Both variables (`result` and `minimum`) are applied in the loop.

Adding an Element to the List

Finding the Smallest Value in the List

Inside the loop the function checks if the currently visited element stores a value smaller than the one which is stores in the `minimum` variable (8th line). If so, then the value is copied to the `minimum` variable and the address of the element that contains the value is stored in the `result` pointer. After the loop terminates the `result` variable stores the address of the element with the smallest number in the whole list.

Adding an Element to the List

Finding a Spot for the Element

```
1  struct list_node *find_next_node(struct list_node *list_pointer,
2                                                    int number)
3  {
4          list_pointer = find_minimum_value_node(list_pointer);
5          struct list_node *start = list_pointer;
6          do {
7              if(list_pointer->data>number)
8                  break;
9              list_pointer = list_pointer->next;
10         } while(list_pointer!=start);
11         return list_pointer;
12 }
```

Adding an Element to the List

Finding a Place to the Element

The `find_next_node()` returns an address of an element of the list before which the new element has to be added. It takes two arguments: the list pointer and the number to be stored in the new element. First, the function calls the `find_minimum_value_node()` function to find the element storing the smallest number in the list. The element becomes the starting point for traversing the list (line no. 5), which takes place in the `do...while` loop. The condition for the loop is defined in the same way as in the previously described function — the loop continues until the list pointer "goes back" to the node from which the loop started (line no. 10). There is however another possible scenario of terminating the loop. It may find an element that stores a number greater than that one to be stored in the new element (7th line). If that happens the loop will be terminated (8th line). Regardless how the loop terminates, the list pointer stores an address of an element before which the new one should be inserted. The address is returned (line no. 11).

Adding an Element to the List

Adding an Element to the List

```
1  void add_node(struct list_node *list_pointer, int number)
2  {
3      if(list_pointer) {
4          struct list_node *new_node = (struct list_node *)
5                                      malloc(sizeof(struct list_node));
6          if(new_node) {
7                  new_node->data = number;
8                  list_pointer =
9                              find_next_node(list_pointer,number);
10                 new_node->next = list_pointer;
11                 new_node->previous = list_pointer->previous;
12                 list_pointer->previous->next = new_node;
13                 list_pointer->previous=new_node;
14         }
15     }
16 }
```

Adding an Element to the List

The `add_node()` actually adds a new element to the list. It takes two arguments: the list pointer and a number that should be stored in the new element. It doesn't return any value, because the result of its performance is visible after the content of the list is displayed on the screen. First the function checks if the list for which the element has to be added is not empty (3th line). If so, it allocates memory for the new element (4th and 5th lines). If the allocation fails the function terminates and the list stays the same as it was before the function call. If however the allocation is successful (6th line) then the number passed to the function is stored in the new element (7th line) and the function finds the address of the element of the list before which the new one should be added. To this end it calls the `find_next_node()` function (lines no. 8 and 9). After the latter terminates the `add_node()` function adds the new element to the list (lines no. 10, 11, 12 and 13). The operation is performed in similar fashion as in the case of adding a new element inside a doubly linked linear list.

Removing an Element From the List

While implementing the operation of removing a single element from the list the following two cases should be considered:

1. the element is removed from a single element circular list,
2. the element is removed from a list consisting of more than one element.

In the first case the list becomes empty after the node is removed. In the second case the list just becomes shorter by one element. The operation shouldn't be performed for an empty list. The state of the list doesn't change only if it is empty or doesn't contain an element for removing.

Removing an Element From the List

```
1  struct list_node *delete_node(struct list_node *list_pointer, int number)
2  {
3      if(list_pointer) {
4          list_pointer = find_next_node(list_pointer,number);
5          list_pointer = list_pointer->previous;
6          if(list_pointer->data == number) {
7              if(list_pointer == list_pointer->next) {
8                  free(list_pointer);
9                  return NULL;
10             }
11             struct list_node *next = list_pointer->next;
12             list_pointer->previous->next = list_pointer->next;
13             list_pointer->next->previous = list_pointer->previous;
14             free(list_pointer);
15             list_pointer=next;
16         }
17     }
18     return list_pointer;
19 }
```

Removing an Element From the List

The `delete_node()` function takes two arguments: the list pointer and a number that should be contained by the element for removing. In case the list has many such elements, removing only one of them is sufficient. The function returns an address of any element that belongs to the list provided the list still is not empty after an element is removed. Otherwise it returns `NULL` value. In the 3th line the `delete_node()` function checks if it is invoked for nonempty list. If so, it tries to locate the element for removing. To the end it calls the `find_next_node()` function (4th line), but the latter function returns the address of the element that stores a number greater than the one passed to the `delete_node()` function. Thus, the latter function "moves back" the list pointer to a previous element and checks if it stores the requested number. In the 7th line the function checks additionally if it's not the only element of the list. To find it out it is enough to check whether any of the pointer fields of the node points to the node. In case of the described function the `next` field was chosen.

Removing an Element From the List

If the condition in the 7th line is satisfied then it means the only node of the list is to be removed. Thus, in the 8th line the function frees the memory for the element and returns the `NULL` value in the 9th line. After that it terminates. If the condition evaluates to false than it means the function removes an element from the list that has at least two elements. Thus, firstly it assigns the address of the next element to be removed to the local pointer named `next` (line no. 12) and then it excludes the element for removing from the list (lines no. 13 and 14). The operation is performed in the same way as when an element is removed from the inside of a doubly linked list. Then the function frees the memory allocated for the element (line no. 15) and the address stored in the `next` pointer is assigned to the list pointer (line no. 16). The assignment is necessary because in the 21th line the function returns the content of the list pointer, thus the pointer has to point to a valid node of the list.

Removing an Element From the List

If the list, which address is passed to the `delete_node()` function, was empty the function would return in the 21th line the `NULL` value. If the list didn't have an element for removing the function would return the same address as it was passed to it while it was invoked.

Printing the Content of the List

```
1  void print_list(struct list_node *list_pointer,
2                                        const unsigned int how_many)
3  {
4      if(list_pointer) {
5          list_pointer = find_minimum_value_node(list_pointer);
6          int i;
7          for(i=0; i<how_many; i++) {
8              struct list_node *start = list_pointer;
9              do {
10                 printf("%d ",list_pointer->data);
11                 list_pointer = list_pointer->next;
12             } while(list_pointer!=start);
13             puts("");
14         }
15     }
16 }
```

Printing the Content of the List

The operation of printing the values of the nodes of the list is implemented in a form of the `print_list()` function. The function takes two arguments and returns no value. The first argument is the list pointer and the second one is a number defining how many times (in separated lines) the content of the list should be displayed on the screen. The parameter by which the number is passed is called `how_many`. The argument passed by the parameter is also called `how_many` and it is the constant, which is defined at the beginning of the program and which value is `2`. After the function checks that the list in not empty (line no. 4), it searches for the node storing the smallest number in the list with the use of the `find_minimum_value_node()` function. It's not necessary, but it makes easier to find out, that the numbers in the list are stored in the ascending order. Next, in the `for` loop the `print_values()` function assigns the address of the element to the local pointer named `start`. The values of the nodes are displayed on the screen inside the `do...while` loop.

Printing the Content of the List

The `start` pointer is used in the loop condition. After the loop terminates the cursor is moved to the next line on the screen by the `puts()` function and depending on the value of the `how_many` parameter, next iteration of the `for` loop begins or the function terminates.

Removing the List

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      if(*list_pointer) {
4          struct list_node *start = *list_pointer;
5          do {
6              struct list_node *next = (*list_pointer)->next;
7              free(*list_pointer);
8              *list_pointer = next;
9          } while(*list_pointer!=start);
10         *list_pointer = NULL;
11     }
12 }
```

Removing the List

The operation of removing the list is implemented similarly as in the cases of the singly linked and doubly lined lists. However, the `remove_list()` function differs from its counterparts for the aforementioned lists by several details: in the 3rd line it checks if the list is not empty, it uses a different kind of loop to delete all nodes of the list, and finally it assigns the `NULL` value to the list pointer (10th line) after the loop terminates. The last activity is necessary, because the list pointer should be empty after the list is destroyed. Since none of the nodes of the list has the `NULL` value stored in its pointer fields, the value has to be assigned directly to the list pointer. In the function the `do...while` is applied for deleting nodes of the list. It is used in a similar way as in the previously described functions. The definition of its condition (9th line) may seem at first incorrect and even dangerous, since the `start` pointer stores an address of an element that is already deleted. But it is a proper expression. The pointer is not dereferenced. The address stored in it is only compared with the address stored in the list pointer.

The main() Function

The First Part

```
1  int main(void)
2  {
3      list_pointer = create_list(1);
4      int i;
5      for(i=2;i<5;i++)
6          add_node(list_pointer,i);
7      for(i=6;i<10;i++)
8          add_node(list_pointer,i);
9      print_list(list_pointer,how_many);
```

The `main()` Function

The First Part

In the first part of the `main()` function a circular doubly linked list with a single element is created. The value of the node is `1` (3rd line). Next, (just like in the cases of previously described lists) nodes of the values ranging from `2` to `4` and from `6` to `9` are added to the list (lines no. 5, 6, 7 and 8). Then the content of the list is displayed twice on the screen according to the value of the `how_many` constant (9th line). By increasing or decreasing the value of the constant before the program compilation the programmer can define how many the values of the nodes of the list are displayed on the screen by a single invocation of the `print_list()` function.

The main() Function

The Second Part

```
1       add_node(list_pointer,0);
2       print_list(list_pointer,how_many);
3       add_node(list_pointer,5);
4       print_list(list_pointer,how_many);
5       add_node(list_pointer,7);
6       print_list(list_pointer,how_many);
7       add_node(list_pointer,10);
8       print_list(list_pointer,how_many);
```

The `main()` Function

The Second Part

In the second part of the `main()` function the nodes storing the numbers `0`, `5`, `7` and `10` are added to the list. After each such an operation the content of the list is displayed twice on the screen.

The main() Function

The Third Part

```
1       list_pointer = delete_node(list_pointer,0);
2       print_list(list_pointer,how_many);
3       list_pointer = delete_node(list_pointer,1);
4       print_list(list_pointer,how_many);
5       list_pointer = delete_node(list_pointer,1);
6       print_list(list_pointer,how_many);
7       list_pointer = delete_node(list_pointer,5);
8       print_list(list_pointer,how_many);
9       list_pointer = delete_node(list_pointer,7);
10      print_list(list_pointer,how_many);
11      list_pointer = delete_node(list_pointer,10);
12      print_list(list_pointer,how_many);
13      remove_list(&list_pointer);
14      return 0;
15  }
```

The `main()` Function

The Third Part

In the third part of the `main()` function the nodes containing numbers `0`, `1`, `1` (again, but this time no element is actually deleted), `5`, `7` (two nodes contain such a number, but only one is removed) and `10` are removed from the list. After each such an operation the content of the list is printed. The lat operation performed on the list is its removal form the computer memory. After that the `main()` function returns `0` and terminates. There are many others way to test functions that implement operations on the circular doubly linked list, but the most basic are performed in the `main()` function of the program.

Summary

The described program doesn't show every functionality of the circular doubly linked list. For example the `print_list()` function doesn't display the content of the list in the reversed order.
Just like the linear lists, the circular lists can be implemented with the use of an array or it can, as it was already mentioned, have a sentinel node. Those lists are applied in operating system in schedulers that utilise the round–robin algorithm and in network subsystems as buffers for transmitted packages. D.E.Knuth in the first volume of "The Art of Computer Programming" describes an algorithm for multiplication of polynomials, where the circular list represents the arguments. Each node of one of such lists stores a single coefficient of one of the polynomials.

Questions

?

The End

Thank You For Your Attention!