Fundamentals of Programming 2

Singly Linked Linear List

Arkadiusz Chrobot

Department of Computer Science

April 2, 2019

Outline

Singly Linked Linear List

A singly linked linear list is an abstract data structure that can store sorted or unsorted data. Unlike an array, the list allows only sequential access to its elements, but a new element can be added in any spot of the list. The operation of removing an element from the list has the same property. There are several kinds of lists. Stacks and queues are special cases of such data structures. The singly linked linear list distinguishes itself from other lists in that it posses a beginning and ending (the first and the last element) and in that it allows traversing its elements only in one direction, at least by default.

Singly Linked Linear List

Singly Linked Linear List The singly linked linear list can be implemented in the form of a dynamically allocated data structure. This implementation of the list is the main subject of the lecture. It is also possible to implement such a list with the use of an array. However, this possibility is only shortly discussed at the end of the lecture. Just like in the cases of other abstract data structures, to implement the list in the form of a dynamically allocated data structure, the definition of the base data type and operations for the data structure are required. In an example program only five basic operation on the singly linked linear list are implemented: creating a list, adding a new element, removing an element, printing the content of the list and removing the whole list. Also an operation of searching for element in the list that stores certain value is implemented in some form.

Implementation

A program that stores natural numbers in the list is used as an example for explaining the details of the singly linked linear list implementation, although the used list can also store integer numbers. The list is also sorted, i.e. the values stored in the list are sorted in ascending order.

Base Type and List Pointer

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  struct list_node {
5      int data;
6      struct list_node *next;
7  } *list_pointer;
```

Base Type and List Pointer

In the program are included the same header files as in the programs from the previous lecture. Also the base type of the list is the same as for queues and stacks with the exception of the name. Just like the case of the other data structures the base type can be adjusted to the current needs of the programmer by adding fields for storing the data. Also a new pointer fields can be added to the data base, but always at least one pointer field has to exist that allows connecting elements of the list. The pointer filed in the last element of the list has always the value of `NULL`. In the presented program the base type has one field of the `int` type that is used for storing a number and another field which is a pointer to the next element. The definition of the type is merged with the definition of the list pointer (line no. 7). The pointer is a global variable and it should hold the address of the first element of the list or be an empty pointer is the list is also empty.

Creating a List

The operation of creating a list, or in other words adding its first element, can be implemented in many ways. In the presented program it is delegated to a separate subroutine, which source code is presented in the next slide.

Creating a List

The `create_list()` Function

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

Creating a List

The `create_list()` Function

The function creates the first element of the array. It takes as an argument the number that should be stored in the element and returns its address, if the operation of creating the element is successful or returns `NULL` otherwise. The memory for the new element is allocated in the lines no. 3 and no. 4. If the allocation is successful, the function initializes fields of the element and returns its address. If the operations fails the function returns the `NULL` value store in the `first` pointer. Please observe, that the `next` filed of the first element is initialized with `NULL` value, because this is the first and at the same time the last element of the list. The value returned by the `create_list()` function should be assigned to the list pointer.

Operation of Adding an Element to the List

It is assumed that the operation of adding a new element to the list is always performed for an existing (i.e. nonempty) list. Additionally, it is required that the list pointer should point to the first element of the list before and after the element is added and the numbers in the list should be sorted ascending. If creating the new element fails the list should stay the same as it was before.

Operation of Adding an Element to the List

Implementation

If the list is to store numbers in a ascending order then the following three cases of adding a new element to the list should be carefully considered:

1. the element is added at the beginning of the list; it becomes the first element of the list,
2. the element is added inside the list,
3. the element is added at the end of the list; it becomes the last element of the list.

The adding of an element to the list is performed by a single function, however with the help of supporting functions that take care of each of the described cased. **Attention! To make it easier to understand the implementation of the operation of adding a new element to the list, all those functions are presented and described in reverse order than their are defined in the program.** The source code of the program is available on the course website.

Operation of Adding an Element to the List

The `add_node()` Function

```
1  struct list_node *add_node(struct list_node *list_pointer, int data)
2  {
3      struct list_node *new_node = (struct list_node *)
4                                          malloc(sizeof(struct list_node));
5      if(list_pointer && new_node) {
6          new_node->data = data;
7          if(list_pointer->data>=data) {
8              return add_at_front(list_pointer, new_node);
9          } else {
10             struct list_node *node= find_spot(list_pointer,data);
11             add_in_middle_or_at_back(node,new_node);
12         }
13     }
14     return list_pointer;
15 }
```

Operation of Adding an Element to the List

The `add_node()` Function

The `add_node()` function is the aforementioned function that adds new element to the list. It takes two arguments. The first one is the list pointer and the second one is the number that should be stored in the new element. The function returns the address of the first element of the list. It is useful when the new element is added at the beginning of the list. In other cases the function returns the same address as it gets through the `list_pointer` parameter. The result of the function should be assigned to the list pointer. The `add_node()` function allocates memory for the new element and then it checks if the operation has been successful and if the list exists (line no. 5). Please note, that a memory leak would be possible if the element was created and the list was empty. In that case the function would return the `NULL` value, but it wouldn't add the element to the list, so its address would be lost.

Operation of Adding an Element to the List

The `add_node()` Function

In the presented program such a situation won't happen, because the function is always called after the `created_list()` function. However, if the function is to be used in other program, a care should be taken to pass the pointer to a nonempty list to it. If the list and the new element exist then the function initializes the `data` field of the new element (line no. 6). The initialization of the `next` field is not necessary, because the appropriate value is assigned to it by the other functions. After the new element is initialized the `add_node()` function has to recognize which of the three cases it should handle. The list to which the element is added is sorted in ascending order, so if the number stored in the new element is lesser or equal to the number stored in the current first element of the list, than the element should be added at the front of the list. This condition is tested in the 7th line. If it is satisfied then the `add_at_front()` function is called and the `add_node()` function returns the address returned by the former function and terminates.

Operation of Adding an Element to the List

The `add_node()` Function

If the condition in the 7th line is not satisfied then there has to be one of the two remaining cases: either the new element should be added inside the list or at its back. It occurs, that both those cases can be handled in the same way. First, the element has to be located in the list *behind which* the new element ought to be added. This task is handled by the `find_spot()` function, which returns the address of such an element. The address is then stored in the `node` pointer. The existence of the list has been already checked before the `find_spot()` function is called, so the function always finds the appropriate element of the list, and the `node` pointer is never an empty pointer. After the element in the list is located the `add_node()` function calls the `add_in_the_middle_or_at_back()` function which actually adds the new element to the list.

Adding a New Element at the Front of the List

The `add_at_front()` Function

```
1  struct list_node *add_at_front(struct list_node *list_pointer,
2                                 struct list_node *new_node)
3  {
4      new_node->next = list_pointer;
5      return new_node;
6  }
```

Adding a New Element at the Front of the List

The `add_at_front()` Function

The `add_at_front()` function is similar to the `push()` function defined for the stack. The former takes two arguments: the address of the first element of the list (the list pointer in other words) and the pointer to the new element. Because the existence of the list and the new element is conformed by the `add_node()` function, the `add_at_front()` function doesn't have to verify it again. However, the latter function shouldn't be used outside the `add_node()` function without checking the value of the pointers that are passed to it. In the 4th line of the function the address of the currently first element of the list is assigned to the `next` field of the new element, and then in the 5th line the function returns the address of the new element (it is now the first element of the list) and terminates.

Adding a New Element Inside and at the Back of the List

The `find_spot()` Function

```
1  struct list_node *find_spot(struct list_node *list_pointer,
2                                                      int data)
3  {
4      struct list_node *previous = NULL;
5      while(list_pointer && list_pointer->data<data) {
6          previous = list_pointer;
7          list_pointer = list_pointer->next;
8      }
9      return previous;
10 }
```

Adding a New Element Inside and at the Back of the List

The `find_spot()` Function

The `find_spot()` function is responsible for locating an element of the list **after** which a new element has to be added and for returning its address. Just like in the case of the `add_at_front()` function, checking the pointer list value is not needed since it has been confirmed by the `add_node()` function. The number stored in the new element is also passed the `find_spot()` function. The elements of the list are traversed in the `while` loop with the use of the list pointer that is passed to the function by a value. Also the `previous` pointer declared in the 4th line is used in the loop. It points the element of the list preceding the element pointed by the list pointer. If the value of the list pointer becomes `NULL`, the `previous` pointer will point to the last element of the list.

Adding a New Element Inside and at the Back of the List

The `find_spot()` Function

According to the condition from the line no. 5 the loop terminates when the list pointer points to an element of the list that stores a number which is equal or greater than the number stored in the new element or when the value of the list pointer is `NULL`. In the latter case the new element should be added at the end of the list, because it stores the biggest number in the list. Summarizing: after the `while` loop terminates, the `list_pointer` variable points to an element of the list **before** which the new element should be added or it has the value of `NULL`. The `previous` pointer points to the element of the list **after** which the new element should be added and its value is returned by the function.

Adding a New Element Inside and at the Back of the List

The `add_in_middle_or_at_back()` Function

```
1  void add_in_middle_or_at_back(struct list_node *node,
2                                struct list_node *new_node)
3  {
4      new_node->next = node->next;
5      node->next = new_node;
6  }
```

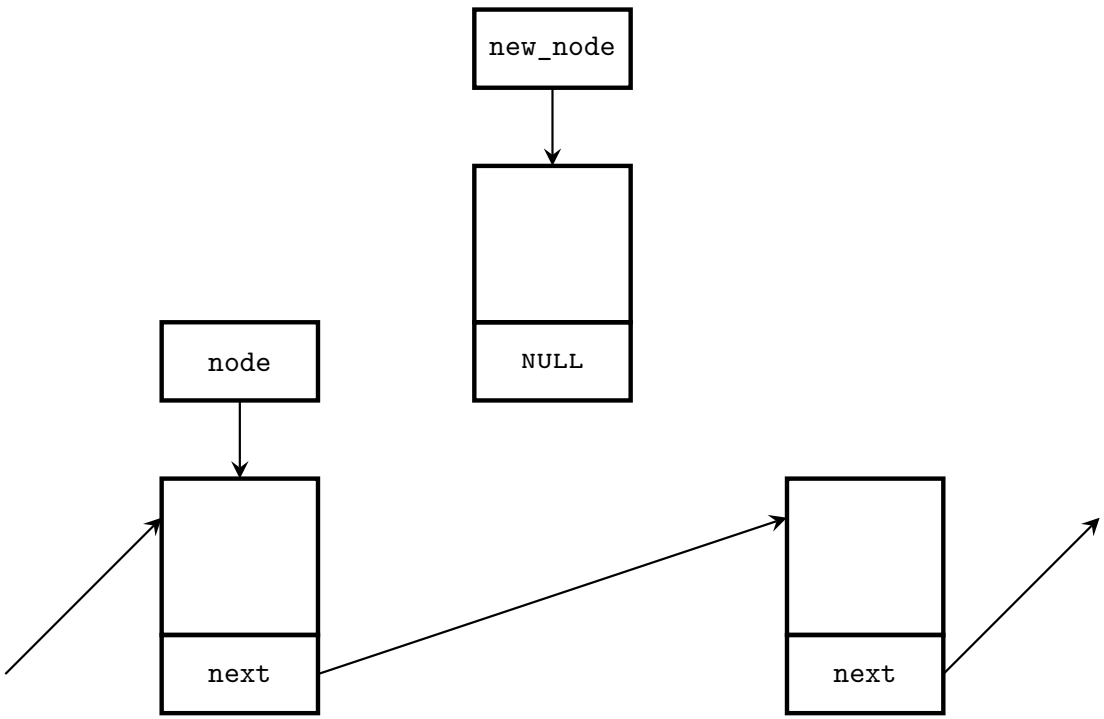
Adding a New Element Inside and at the Back of the List

The `add_in_middle_or_at_back()` Function

The `add_in_middle_or_at_back()` function job is to add a new element at the end or inside the list. After it finishes the list should be consistent, i.e. the list should have a new element and all the elements should be linked together. By the first parameter is passed the address of the element after which the new one should be added. The address of the new element is passed by the second parameter. Because the function is called by the `add_node()` function there is no need for verifying those pointers. In the 4th line of the function the address of the element of the list which succeeds the one pointed by the node parameter is assigned to the `next` field of the new element. In the 5th line the address of the new element is stored in the `next` field of the list's element pointed by the `node` parameter. After the assignments are preformed the new element becomes a part of the list. The lines no. 4 and no. 5 cannot switch places. The next slide contains an animation that illustrates the described activities.

Adding a New Element Inside and at the Back of the List

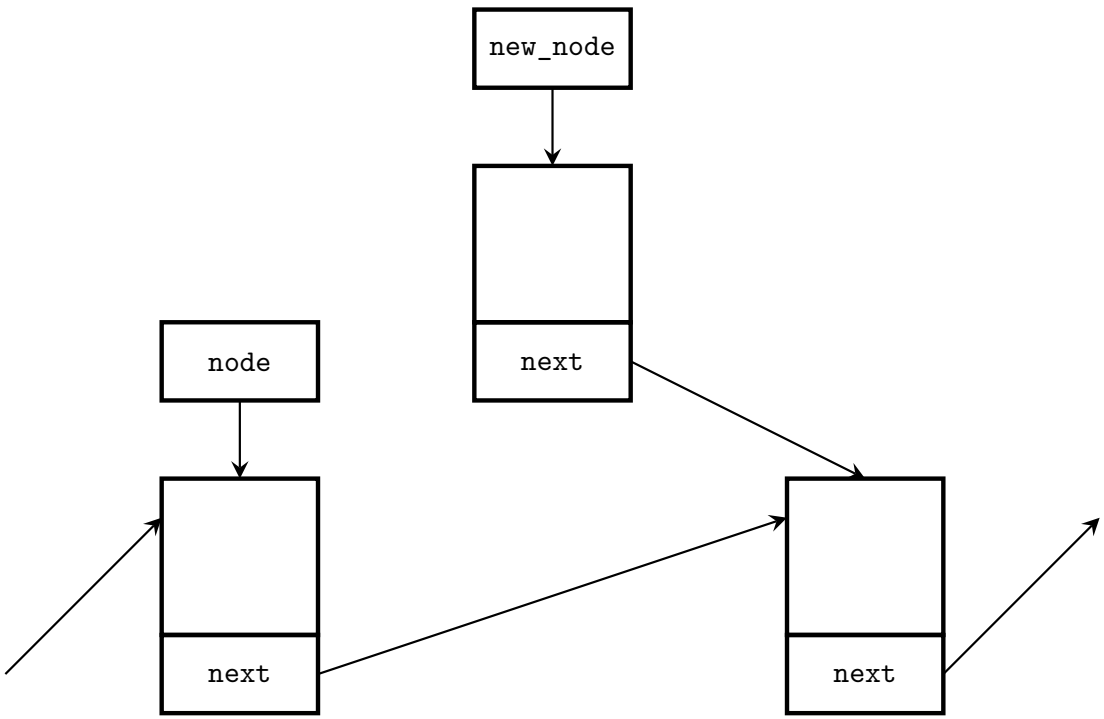
The `add_in_middle_or_at_back()` Function



Before the line no. 4 of the `add_in_middle_or_at_back()` function is performed

Adding a New Element Inside and at the Back of the List

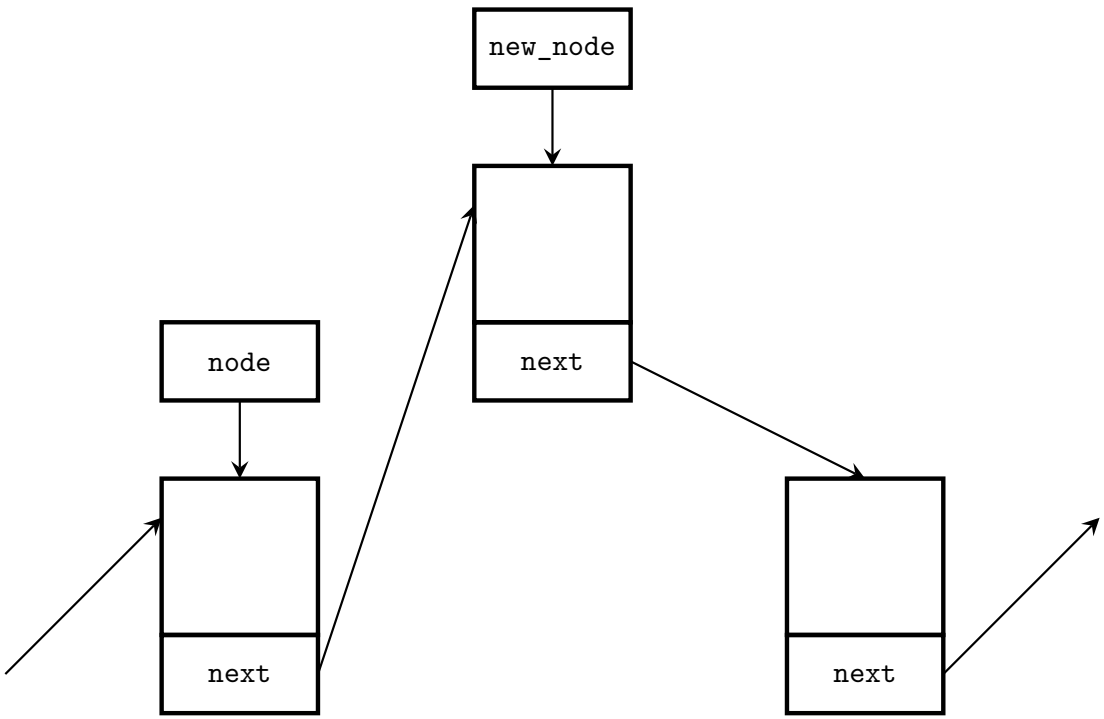
The `add_in_middle_or_at_back()` Function



After the line no. 4 of the `add_in_middle_or_at_back()` function is performed

Adding a New Element Inside and at the Back of the List

The `add_in_middle_or_at_back()` Function



After the line no. 5 of the `add_in_middle_or_at_back()` function is performed

Adding a New Element Inside and at the Back of the List

The `add_in_middle_or_at_back()` Function

Let's consider the behaviours of the `add_in_middle_or_at_back()` when the element pointed by the `node` pointer is the last element of the list. In that case, in 4th line of the function, the `NULL` value is assigned to the `next` filed of the new element, because that value is stored in the `next` field of the current last element of the list pointed by the `node` variable. In the 5th line the address of the element which is added to the list is stored in the `next` filed of the element of the list pointed by the `node` pointer. Thus the new element becomes the last element of the list and it satisfies the most important condition for such an element — its `next` field has the `NULL` value.

Adding an Element to the List — Summary

The `add_node()` function could be written without partitioning its source coded into smaller functions, but its definition would be probably longer and less readable. If increasing the efficiency of the function was required the helper functions would be defined with the use of `static inline` keywords. In that case the compiler would likely handle them in similar fashion as the preprocessor handles macros. The operation of adding an element to the list can be implemented in other way then presented. For example the list pointer could be passed to the `add_node()` function with the use of pointer to a pointer.

Operation of Removing an Element from the List

The operation of removing a single element from a singly linked linear list, similarly to the operation of adding an element to the list, has to be performed on a list that has at least one element. An element will be removed if it contains a specified number. If there is more than one element on the list that stores the number then the first which will be found will also be removed. After the operation is performed the list should have one element less, but it still should be consistent or be empty.

Operation of Removing an Element from the List

Implementation

When implementing the operation of removing a single element from the list, the following four cases should be considered:

1. the removed element is the first element of the list,
2. the removed element is inside of the list,
3. the removed element is the last element of the list,
4. there is no element in the list that should be removed; the list should stay the same as it was.

Similarly as in the case of adding, removing of an element is performed by a single function which uses several helper functions. **The functions are presented and described in reversed order than they are defined in the program.**

Operation of Removing an Element from the List

The `delete_node()` Function

```
1  struct list_node *delete_node(struct list_node *list_pointer,
2                                                    int data)
3  {
4      if(list_pointer) {
5          if(list_pointer->data==data)
6              return delete_at_front(list_pointer);
7          else {
8              struct list_node *previous =
9                              find_previous_node(list_pointer,data);
10             delete_middle_or_last_node(previous);
11         }
12     }
13     return list_pointer;
14 }
```

Operation of Removing an Element from the List

The `delete_node()` Function

The `delete_node()` function is responsible for removing a single element from a list. It takes the list pointer and the number that should be stored in the removed element as arguments. The function returns the address of the first element of the list, which should be assigned to the list pointer. If the function removes the first element of the list it will return the address of an element that becomes the new first element in the list, otherwise it will return the value passed to it by its first parameter. In the 4th line the function checks if the list exists. If so, it tries to locate the element that should be removed and if it is found, the function removes it. In the 5th line the function tests if the first element of the list should be removed. To the end it compares the number passed to the function by its second parameter with the number stored in the `data` field of the element.

Operation of Removing an Element from the List

The `delete_node()` Function

If the condition is satisfied, the `delete_node()` function invokes the `delete_at_front()` function, which performs the operation of removing the first element of the list and returns the address of the new first element of the list which in turn is returned by the `delete_node()` function which also terminates. Otherwise, the latter function has to find the element, which should be removed from the list, and so it calls the `find_previous_node()` function which returns the address of the element **preceding** the element that should be removed. It can however happen, that there is no element to be removed on the list. In that case the `find_previous_node()` function returns the address of the last element of the list. All other cases of removing an element from the list are handled by the `delete_middle_or_last_node()` function, which is described in the next slides.

Removing from the Beginning of the List

The `delete_at_front()` Function

```
1  struct list_node *delete_at_front(struct list_node *list_pointer)
2  {
3      struct list_node *next = list_pointer->next;
4      free(list_pointer);
5      return next;
6  }
```

Removing From the Beginning of the List

The `delete_at_front()` Function

The first element of the list is removed by the `delete_at_front()` function. The function takes as arguments the address of the first element of the list and it returns the address of the element that was second in the list, before the function was invoked. The latter element becomes the first one in the list after the function terminates. The result of the function is returned to the `delete_node()` function and eventually assigned to the list pointer. The behaviour of the `delete_at_front()` function is similar to the behaviour of the `pop()` function defined for a stack. Because the former function in invoked by the `delete_node()` function, no verification of the list pointer is required. In the 3rd line the function stores the address of the second element in the list in a local pointer named `next`. Then it frees memory allocated for the first element of the list (line no. 4) and terminates returning the address of the new first element of the list (line no. 5).

Removing From the Middle and at the End of the List

The `find_previous_node()` Function

```
1  struct list_node *find_previous_node
2                          (struct list_node *list_pointer, int data)
3  {
4      struct list_node *previous = NULL;
5      while(list_pointer && list_pointer->data!=data) {
6          previous=list_pointer;
7          list_pointer=list_pointer->next;
8      }
9      return previous;
10 }
```

Removing From the Middle and at the End of the List

The `find_previous_node()` Function

The `find_previous_node()` function is responsible for locating an element in the list that precedes the element that has to be removed. If there is no element in the list to be removed, the function returns the address of the last element of the list. Please note the similarity of the definition of the function to the definition of the `find_spot()` function. The element is located inside the `while` loop. In traversing the list two pointers are used: the `list_pointer` and the `previous`, which plays the same role as its counterpart in the `find_spot()` function. The loop stops (refer to the line no. 5) when list pointer value is `NULL` or points element that has to be removed, because of the value of its `data` field. However, the function returns the address of the element that **precedes** the removed one. Please observe that the expressions in the condition in the 5th line cannot change their places. Otherwise the function would reference the `data` field of an element before checking if the element exists. It could cause the program to fail.

Removing From the Middle and at the End of the List

The `delete_middle_or_last_node()` Function

```
1  void delete_middle_or_last_node(struct list_node *previous)
2  {
3      struct list_node *node = previous->next;
4      if(node) {
5          previous->next = node->next;
6          free(node);
7      }
8  }
```

Removing From the Middle and at the End of the List

The `delete_middle_or_last_node()` Function

The `delete_middle_or_last_node()` function, contrary to what its name suggests, also handles the last possible case of removing an element from a list — when there is no element to be removed. In that case the function takes no action and the list stays the same as it was. The function takes the address of the element preceding the element to be removed from the list as an argument. Similarly as in the case of previously described helper functions, there is no need for verifying the pointer, because the `find_previous_node()` function never returns `NULL`. In the 3rd line the function stores the address of the element pointed by the `previous` parameter in the locally defined `node` pointer. If the pointer in not empty, which is verified in the 4th line, it means that an element exists that should be removed from the list. It can be the last element of the list or an element inside the list. It occurs that both cases can be handled by the same statements.

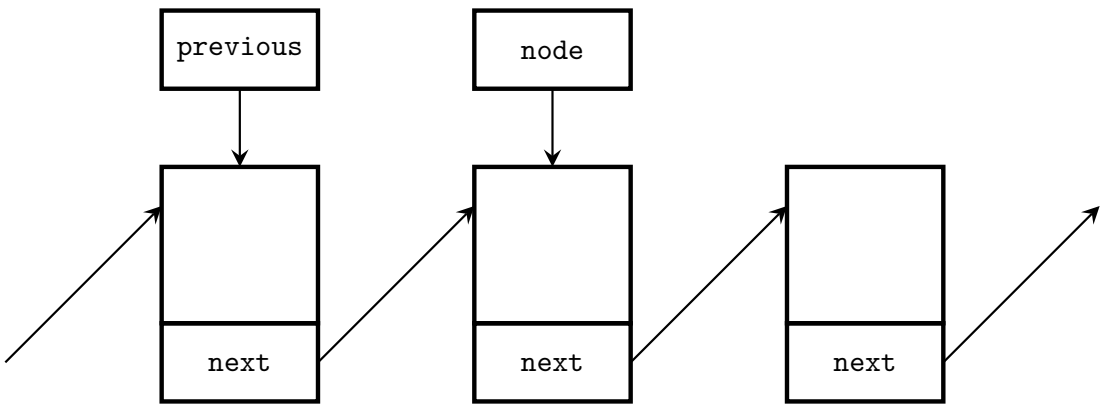
Removing From the Middle and at the End of the List

The `delete_middle_or_last_node()` Function

In the 5th line the address from the **`next`** field of the element pointed by the **`node`** pointer is assigned to the **`next`** field of the element pointed by the **`previous`** pointer. If the former field is empty then the removed element is the last element of the list and after it is removed the element pointed by the **`previous`** pointer becomes the last one in the list. Thanks to the assignment in the 5th line the **`next`** field gets the value of `NULL` which is required for the last element of the list. If however, the element pointed by the **`node`** pointer is not the last one in the list, then performing the statement from the 5th line will unlink it from the rest of the list. In the 6th line, the memory allocated for the element pointed by the **`node`** pointer is freed and the function terminates. The next slide contains an animation that illustrates the operation of removing an element from the inside of the list.

Removing From the Middle and at the End of the List

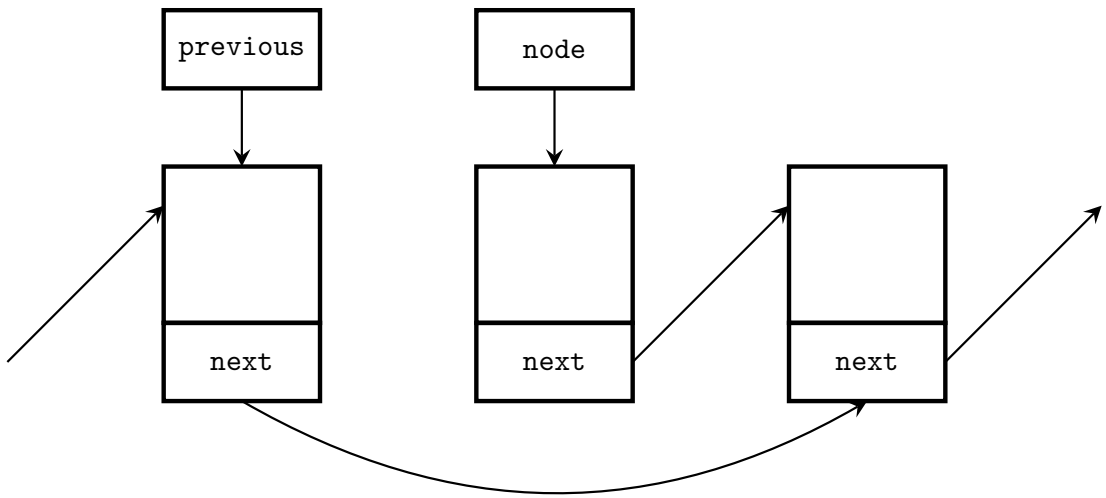
The `delete_middle_or_last_node()` Function



Before performing the 5th line of the `delete_middle_or_last_node()` function

Removing From the Middle and at the End of the List

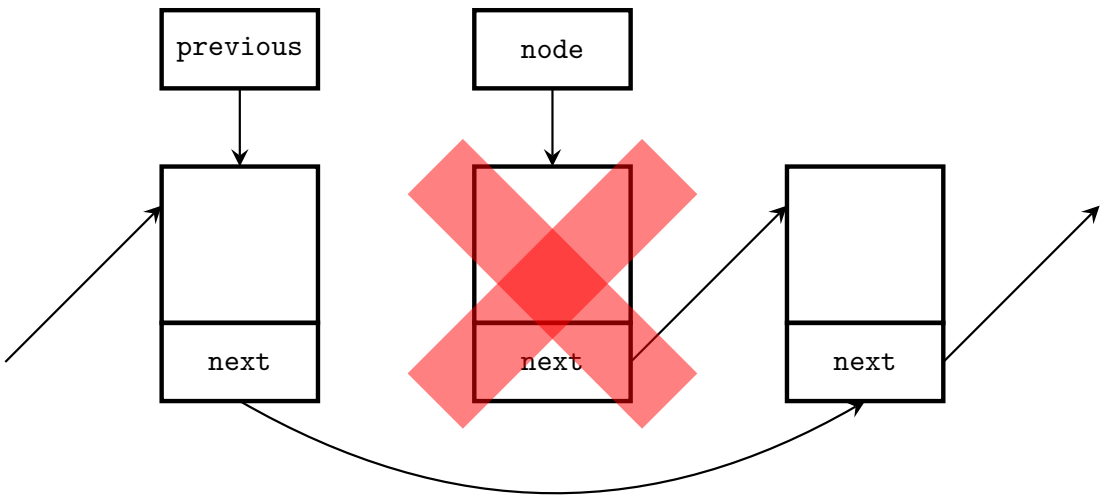
The `delete_middle_or_last_node()` Function



After performing the 5th line of the `delete_middle_or_last_node()` function

Removing From the Middle and at the End of the List

The `delete_middle_or_last_node()` Function



After performing the 6th line of the `delete_middle_or_last_node()` function

Removing an Element From the List — Summary

Similarly as the `add_node()` function the `delete_node()` function can be implemented without defining the helper functions, but it is also likely it definition would be less legible. The effectiveness of the function can be improved by defining all helper functions with the use of the `static inline` keywords. Also it is possible to implement the function differently than it is presented in the lecture.

Operation of Displaying the Content of the List

Printing of all numbers stored in the elements of the list is implemented in the same way as in the case of the FIFO queue, hence the `print_list()` function is not described in details.

Operation of Displaying the Content of the List

The `print_list()` Function

```
1  void print_list(struct list_node *list_pointer)
2  {
3      while(list_pointer) {
4          printf("%d ",list_pointer->data);
5          list_pointer=list_pointer->next;
6      }
7      puts("");
8  }
```

Operation of Removing the List

The removal of the list consists of removing all its elements, or of freeing the memory allocated for each of the elements, starting from the first one. After the operation is finished the list pointer should have the value of `NULL`. The operation is implemented in the form of the `remove_list()` function.

Operation of Removing the List

The `remove_list()` Function

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      while(*list_pointer) {
4          struct list_node *next = (*list_pointer)->next;
5          free(*list_pointer);
6          *list_pointer = next;
7      }
8  }
```

Operation of Removing the List

The `remove_list()` Function

The `remove_list()` function takes the list pointer as its argument. In the `while` loop all the elements of the list are successively removed until the list is empty and the list pointer has the value of `NULL`. Please note that the 4th, 5th and 6th lines of the function are very similar to the statements in the `pop()` function that was introduced in the lecture on the stack. The elements of the list are destroyed starting from the first one and finishing with the last one. Please note the assignment in the 4th line. The parentheses on the right side of the operator are necessary to force the precedence of the operators — the dereference of the list pointer has to take the place before the `next` field of the element pointed by this pointer can be accessed. Without the parentheses the operators could be performed in the wrong order, which is signaled as an error by the compiler.

The `main()` Function

In the `main()` function all the functions that implement the basic operations on the list are invoked. To verify the correctness of their behaviour such numbers are passed to the functions that all of the following cases are checked:

- adding an element at the beginning of the list,
- adding an element inside the list,
- adding an element at the end of the list,
- removing an element from the beginning of the list,
- removing an element from the middle of the list,
- removing an element from the end of the list,
- removing a nonexistent element from the list.

The main() Function

First Part

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

The `main()` Function

First Part

In the first part of the `main()` function the list is created. It initially consists of a single element which stores the `1` number. Next the elements of the values from `2` to `4` and from `6` to `9` are added to the list. After the element are added, the list is printed on the screen.

The main() Function

Second Part

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

The `main()` Function

Second Part

In the second part of the `main()` function the elements of the values `0`, `5`, `7` and `10` are added to the list. The first one is added at the beginning of the list, the second in the middle of the list, the third also in the middle of the list, before an element of the same value and the last one at the end of the list. After each addition the content of the list is displayed on the screen, so the user can make sure that the operations are performed correctly.

The main() Function

Third Part

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

The `main()` Function

Third Part

In the third part of the `main()` function the elements that store values `0` (at the beginning of the list), `1` (at the new beginning of the list), `1` (doesn't exist any more on the list), `5` (in the middle of the list) and `10` (at the end of the list) are successively removed from the list. After each such an operation is performed the content of the list is displayed on the screen. Finally, the list is removed with the use of the `remove_list()` function and the `main()` function terminates.

Applications

The lists implemented as singly liked or doubly linked lists, which will be discussed in the future lectures, have many applications. Typically they are used in the operating systems. The Linux kernel applies them frequently and the Linux programmers have provided a default implementation of a list that is used in many parts of the kernel. The majority of the contemporary programming languages provides predefined implementations of such data structures. In some of them the lists are implemented as a part of the language standard library (Java, for example) or as the integral part of the language (Python, for example).

Summary

Singly linked linear lists can be implemented with the use of sentinels, just like the queues. In that case, when the program starts, one or two such elements are created. Thanks to them the functions that implement the operations of adding and removing elements of the list don't have to check some of the conditions. The lists can also be implemented with the use of multidimensional arrays. The first row of the array stores the values of the elements of the list and the second one stores the values of the indices which correspond to the addresses of the next element of the list. Unfortunately, the capacity of the list is limited and the operation of adding a new element is complicated to implement, because it requires of coping some of the elements of the array. Such an activity may also be required by the operation removing an element from the list. Both operations, however, can be implemented differently, but it requires defining, that a specific value of the index in the second row of the array indicates a removed element of the list.

Summary

The removed elements can be latter used for inserting new elements to the list. There has to be also defined an index that would be the counterpart of the `NULL` value. The singly linked linear list can also be implemented with the use of a linear array. All elements with the even indices would be the counterparts of the `next` pointer fields. Both implementations get complicated if the list should store the values of more complex data types. However, this form of implementing the list is necessary in case of older programming languages that do not support dynamical allocation and deallocation of the memory or in case where the program is developed for a computer system of limited memory resources.

Questions

?

The End

Thank You For Your Attention!