Fundamentals of Programming 2

Stack And Queue

Arkadiusz Chrobot

Department of Information Systems

April 11, 2025

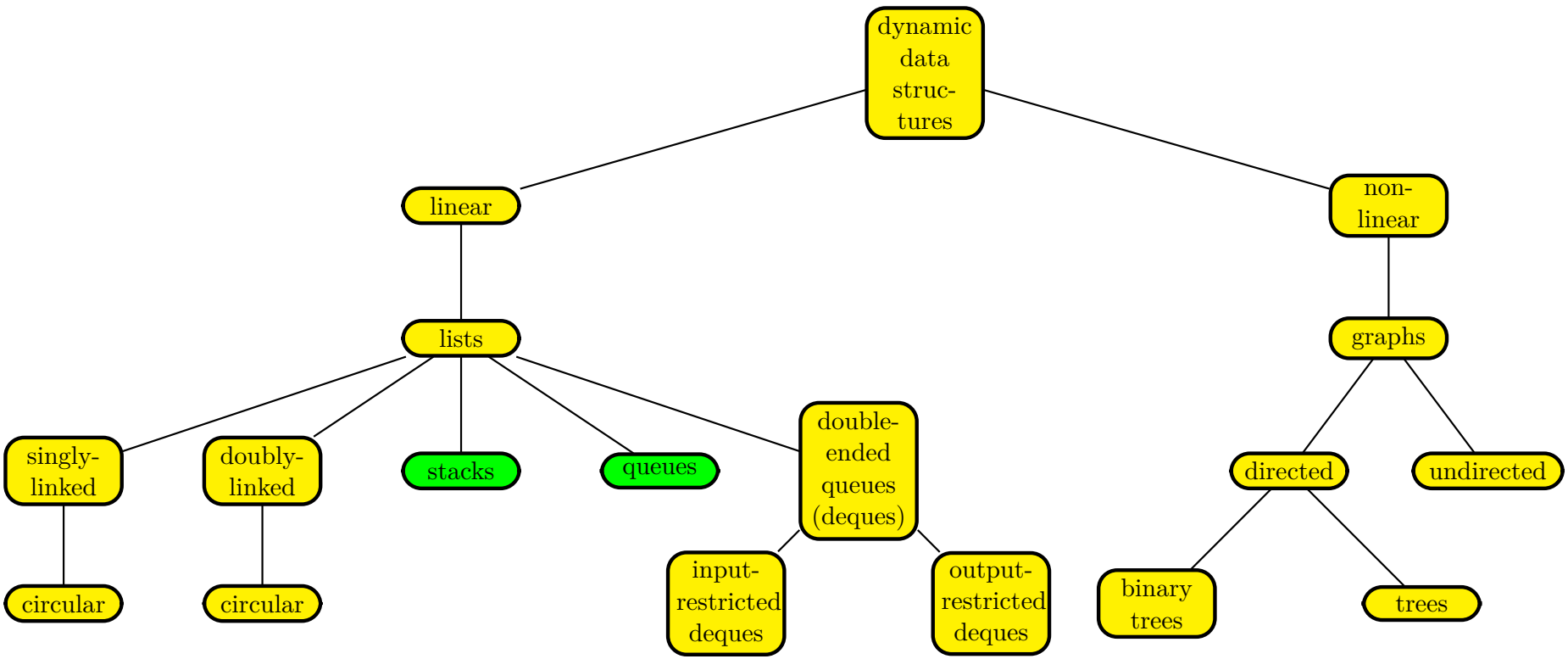
Outline

1. Data Structures
2. Stack — Introduction
3. Stack — Implementation
4. Reverse Polish Notation
5. Queue
6. Summary

Data Structures

The dynamic allocation and deallocation of memory allows programmers to build data structures that are not a part of the language standard. Most of them are based on structures and pointers, but they also require specific operations that are implemented using functions.
The next slide shows a classification of some of the data structures that can be created that way. In this lecture, the stack and queue will be described.
The concept of stack has already been discussed in the lecture on the recursion. It is a data structure where the order of adding and removing elements (called nodes) is defined as *Last In First Out* (LIFO), meaning they are retrieved from the stack in reverse.
The queue is a similar data structure, but its nodes are stored in the *First In First Out* (FIFO) order.

Classification of Data Structures



Classification of data structures

Stack — Introduction

The stack can be implemented in several ways. One of them is a dynamic data structure whose nodes, that store data, are linked together with the use of pointers. It is also a special case of another data structure called a *list*. To better understand what the stack is, let's assume, that a list is a collection of linked together nodes with two ends. The stack is a kind of list in which operations of adding and removing elements can be applied to only one of its ends. Usually the stack is depicted as a vertical list, with one end called the *top*.

Stack — Implementation

Implementing the stack requires not only defining the type of its nodes, but also specifying operations that can be applied to this data structure. The first part can be accomplished in the C language using structures, and the second with the use of functions.
In this lecture it is assumed that the stack should store, in its nodes, numbers of the `double` type, however that can be changed according to the programmer's need. Generally, a single node of the stack can even store more than one value.

Stack — Implementation

Data Type For Stack Node

```
struct stack_node {
        double number;
        struct stack_node *next;
};
```

Stack — Implementation

The previous slide contains a definition of a type of structure which also specifies the type of single stack node. It has two fields. The first one stores data, in this case a number of the `double` type. The second member of the node type is a pointer. There are stacks with elements containing more than one pointer field. However, in any dynamically allocated stack, there should be at least one such a pointer field in each node. Please notice the type of the pointer field. It is the same as the type of the structure that contains it. It means that the structure is *recursive* and that the pointer can point to another structure of the same type as the one it is contained in. In other words, thanks to this pointer, the nodes of the stack can be linked together.

Stack — Implementation

The nodes of the stack are linked together using pointer fields. However, to perform an operation on the stack, the program has to know where the top of the stack is. Therefore, a separate pointer, local or global, is needed to store the address of the top of the stack. This pointer can have any name, but usually it is referred to as a *stack pointer*.

The only thing now missing from the implementation of the stack are the operations. The most basic of these are: adding a new element to the stack, called *push*, and removing an element from the stack, called *pop*. Both operations are performed on the top of the stack. An optional operation is retrieving the data stored in the top node of the stack, called *peek*. All three operations are defined in the lecture.

Stack — Implementation

The `push()` Function

The *push* operation is implemented in a form of `push()` function. The working of this function should satisfy the following conditions (assertions):

1. Before the function is performed the stack pointer has to point the top element of the stack or be an empty pointer — in the latter case the stack is also empty.
2. The function as a result should return an address that either is the same as the one stored in the stack pointer — in that case adding a new element to the stack has failed — or it is an address of a new element on the top of the stack — in that case the operation has been successful.

Stack — Implementation

The `push()` Function

```
1  struct stack_node *push(struct stack_node *top, double number)
2  {
3          struct stack_node *new_node = (struct stack_node *)
4                  malloc(sizeof(struct stack_node));
5          if(new_node!=NULL) {
6                  new_node->number = number;
7                  new_node->next = top;
8                  top = new_node;
9          }
10         return top;
11 }
```

Warning! The line numbers are not a part of the source code. They are introduced to simplify describing of the function's code.

Stack — Implementation

The `push()` Function

The `push()` function takes two arguments which are passed by its parameters. The first one is the stack pointer and the second is a number that will be stored in a new node of the stack. First, the function allocates memory for the new node (lines no. 3 and 4). What happens next depends on the result of this operation. If it fails then the value of the `new_node` pointer will be `NULL` and the function will return the unchanged value of the `top` pointer. Otherwise, the `new_node` pointer will store the address of the new element. The number passed by the `number` parameter is assigned to the `number` field of the new element (line no. 6). In the 7^{th} line, the address currently stored in the `top` pointer in assigned to the `next` filed of the new node. Thus, the new node is linked to the rest of the stack and becomes a new top of the stack. Therefore, its address is assigned to the `top` pointer in the 8^{th} line. After that the function returns the address of a new top of the stack and terminates.

Stack — Implementation

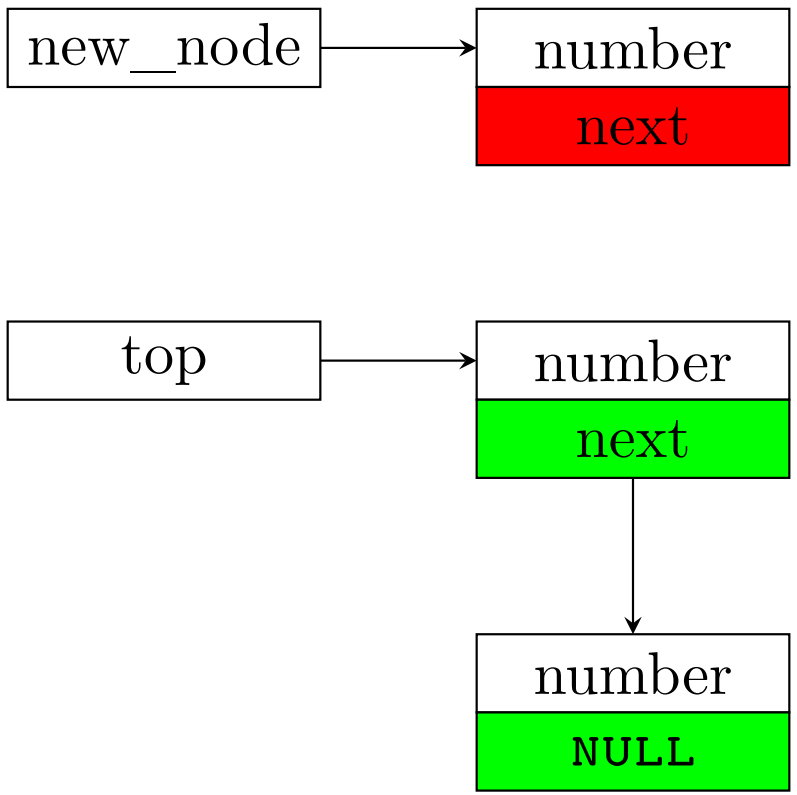
The `push()` Function

The `push()` function can be implemented in many ways. For example, the stack pointer may be passed by a parameter of the pointer to a pointer type, which is then modified in the function body. Similar solution is discussed in more details in the `pop()` function description.

The next slides illustrate successful adding of a new node to a non-empty stack which has two elements. Please notice, that the `next` field of the new node is initially marked in a red color, meaning that it is an incorrect pointer (a wild pointer).

Stack — Implementation

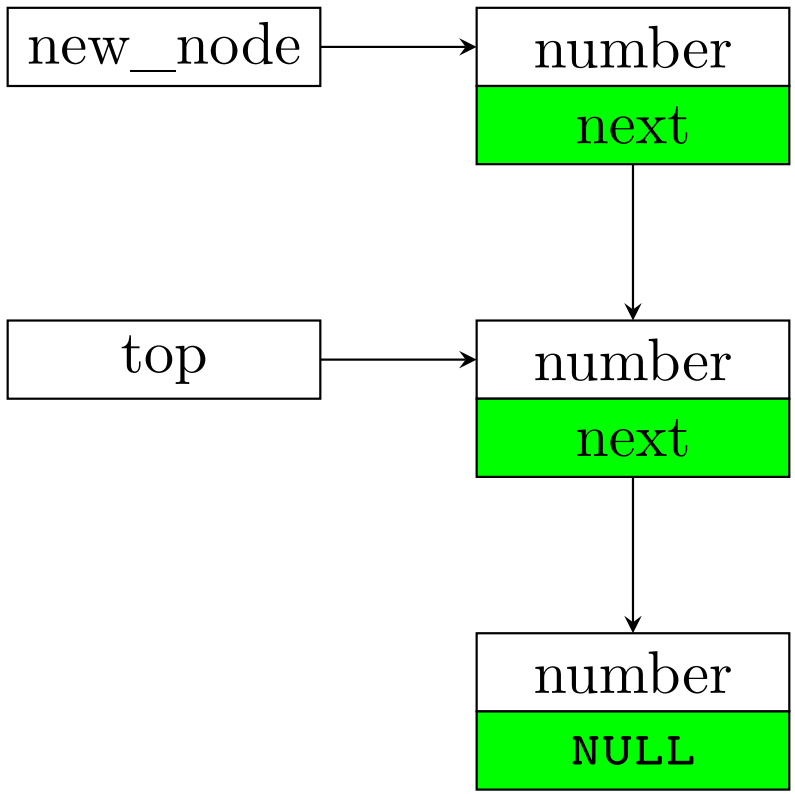
The `push()` Function



Before the 7^{th} line of the `push()` function is performed.

Stack — Implementation

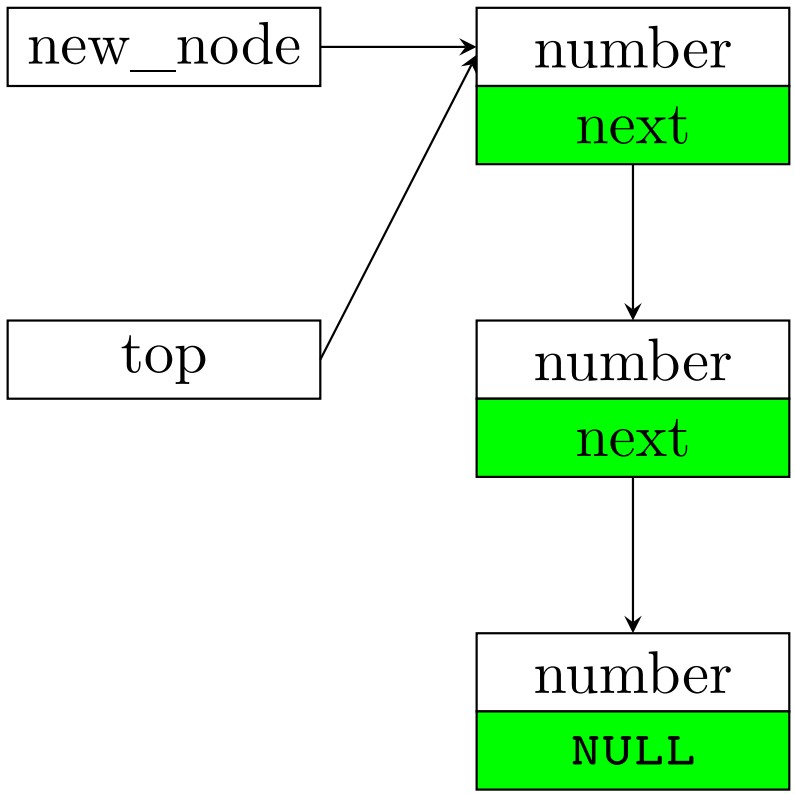
The `push()` Function



Before the 8^{th} line of the `push()` function is performed.

Stack — Implementation

The `push()` Function



Before the 9^{th} line of the `push()` function is performed.

Stack — Implementation

The `push()` Function

Please notice the value of the `next` field in the *bottom* node of the stack. It is `NULL`, meaning that the node is the last element of the stack. There are no other stack nodes behind it. Let's check if the `push()` function assures that the `next` field in the bottom node of the stack always gets the `NULL` value. It turns out, that it happens only when the function is given a stack pointer with the `NULL` value when it is called for the first time. In such a case the `NULL` value is assigned to the `next` field of the first and only element of the stack. However, if in such a case a wild pointer is passed to the function, then its value will be assigned to the `next` field. It is a dangerous situation from the program point of view, because it will be unable to locate the end of the stack. Thus, programmers should always take care of passing an empty stack pointer to the `push()` function when it is invoked to add the first node to the stack. This is especially important when the stack pointer is a local variable.

Stack — Implementation

The `pop()` Function

The *pop* operation is implemented in the form of `pop()` function. Similarly, as in the case of the `push()` function the working of the `pop()` function should satisfy the following assertions:

1. Before the function is performed the stack pointer should point to the top node of a stack, or be an empty pointer.
2. After the function is performed, the stack pointer should point to the top element of the stack, which is one node shorter, or be an empty pointer.

Stack — Implementation

The `pop()` Function

```
1  double pop(struct stack_node **top)
2  {
3          double result = 0.0;
4          if(*top) {
5                  result = (*top)->number;
6                  struct stack_node *temporary = (*top)->next;
7                  free(*top);
8                  *top = temporary;
9          }
10         return result;
11 }
```

Stack — Implementation

The `pop()` Function

The `pop()` function has only one parameter which is a *pointer to a pointer*. By this parameter the address of the stack pointer is passed to this function. Using it is necessary because the function needs to modify the stack pointer and it is not possible to return its new value, because the `pop()` function has to return the number stored in the removed stack node. In this case the pointer to a pointer as a parameter is the best option.

Stack — Implementation

The `pop()` Function

The `pop()` function has a local variable (`result`) of the `double` type whose initial value is set to `0.0`. If the stack is empty then the function will return such a value and terminate. The state of the stack (empty or not) is verified in the 4^{th} line of the function. The `*top` condition is a shorter form of the `*top!=NULL` expression. If it's true the function assigns the number from the current top node of the stack to the `result` variable (line no. 5) and stores the address of a next node in the `temporary` pointer (line no. 6). The address is taken from the `next` field of the stack current top node. Then the top element is removed (line no. 7). Now, the stack pointer has an incorrect value — it doesn't point to the top of the stack. To fix it, in the 8^{th} line the value of `temporary` pointer is assigned to the stack pointer. Next, the function returns the value of the `result` variable and terminates.

Stack — Implementation

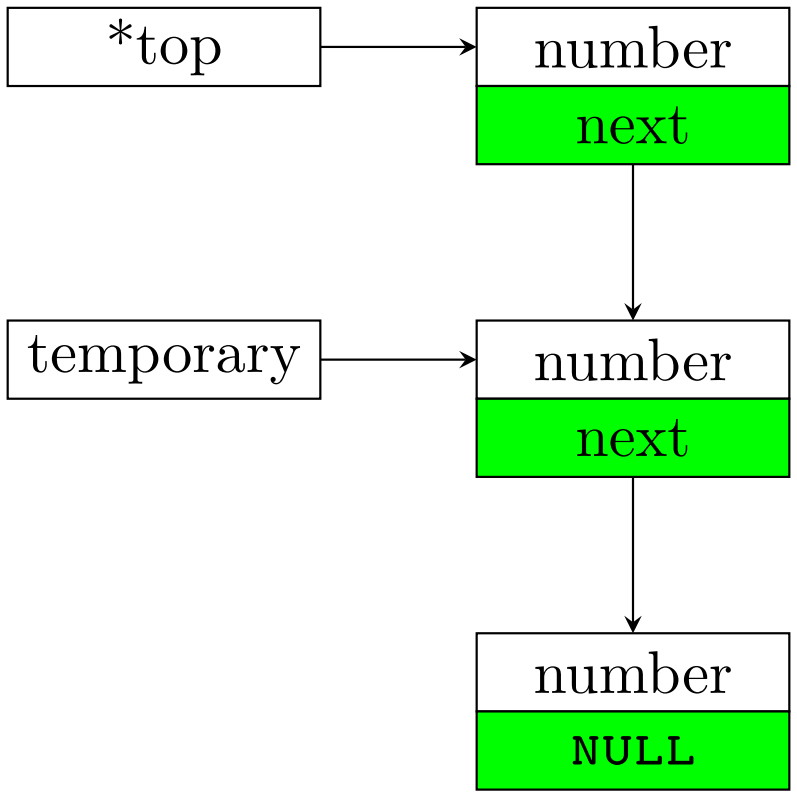
The `pop()` Function

Please observe, that the `pop()` function removes correctly also the last (the bottom) node of the stack. Its `next` pointer field value is `NULL` and such a value is assigned to the `temporary` pointer when the 6^{th} line of the function is performed on a stack with only one node. After the 8^{th} line is executed also the stack pointer gets such a value. This is an expected result, since the stack should become empty after its only node is removed.

The next slides shows the working of the `pop()` function when it removes a top node from a stack that initially has three of them. Contrary to what the slide suggests, the content of the removed element doesn't vanish after the `free()` function is called, nor the pointer that points to it becomes empty. Nonetheless, the node should not be accessed any more. Also the pointer should not be used until a new address is stored in it. The reasons for such restrictions were explained in the first lecture.

Stack — Implementation

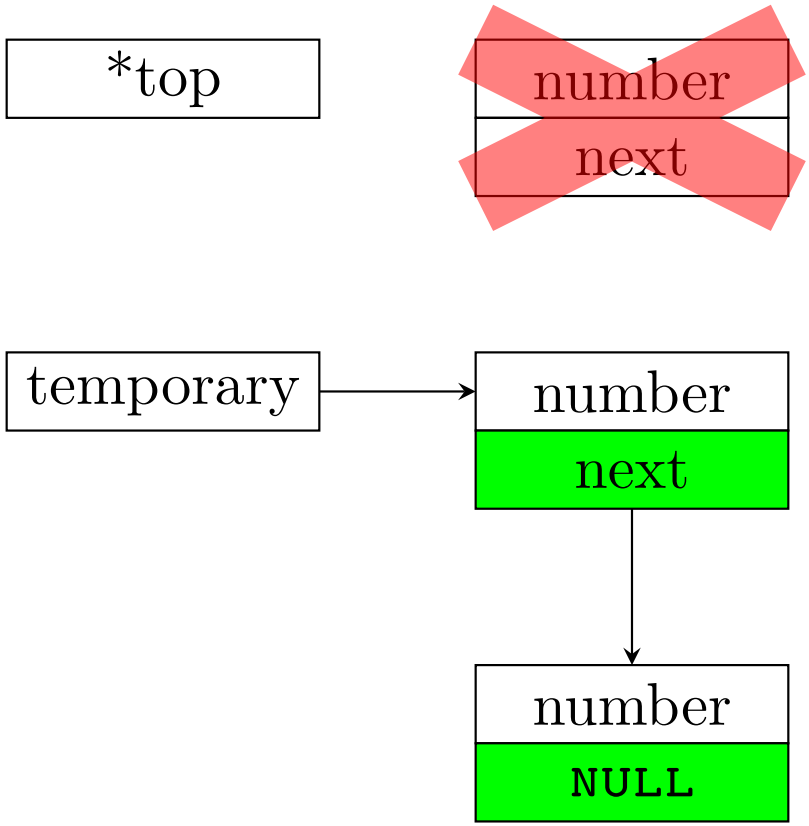
The `pop()` Function



After the 6^{th} line of the `pop()` function is performed.

Stack — Implementation

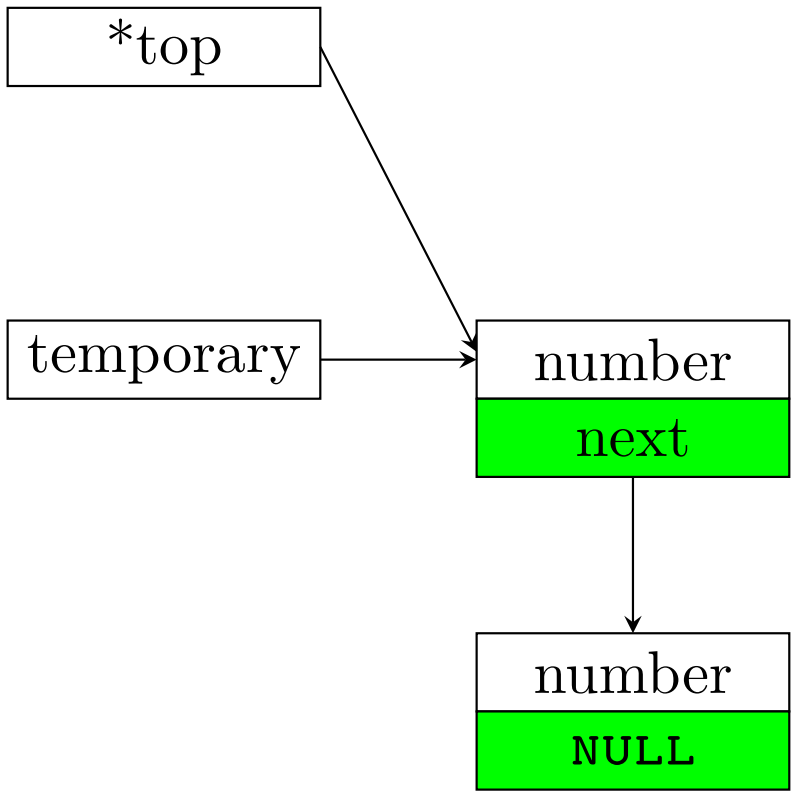
The `pop()` Function



After the 7^{th} line of the `pop()` function is performed.

Stack — Implementation

The `pop()` Function



After the 8^{th} line of the `pop()` function is performed.

Stack — Implementation

The `peek()` Function — Optional

```
1  double peek(struct stack_node *top)
2  {
3          if(top)
4                  return top->number;
5          else {
6                  fprintf(stderr,"The stack is empty.\n");
7                  return 0.0;
8          }
9  }
```

Stack — Implementation

The `peek()` Function — Optional

The *peek* operation is optional. It doesn't have to be defined in every stack implementation. Nonetheless, it is presented in this lecture in the form of `peek()` function. Its definition in relatively simple. The stack pointer is passed to the function with the use of its parameter. If it is not empty (the condition `top` is shorter form of the expression `top!=NULL`), then the function returns the number stored in the top node of the stack. Otherwise, it prints a message on the screen, informing the user that the stack is empty and returns the same value as the `pop()` function in the same case.

Memory Leaks

The implementations of dynamical data structures are prone to serious errors. Incorrectly linked elements of a stack or similar structure are one of the examples. Let's assume that some overzealous programmer decides to zero out the `top` parameter at the beginning of the `push()` function. Such a mistake results in the lack of connections between nodes of the stack. Moreover, aside from the last node, non-other is pointed by any pointer. These nodes cannot be deallocated. The areas of the heap that are allocated to them are lost until the program exits. In the Computer Science jargon such a mistake is called a *memory leak*. In the worst case it can lead to exhaustion of the space in the heap. The first defence against memory leaks is to avoid them by carefully analysing implementations of all operations performed on the data structure. There exist also software tools like debuggers and dedicated libraries that make detecting of such errors easier. Unfortunately, they are not part of the C language standard, because their internal working depends on the used computer and operating system.

Stack — Applications

Evaluating RPN Expressions

The stack is applied for evaluating arithmetic expressions in the *Reversed Polish Notation* (RPN) also called the *postfix* notation. It was invented by an Australian computer scientist and philosopher Charles Hamblin and it is based on the *Polish Notation* (PN) also called the *prefix* notation proposed by a Polish mathematician and philosopher Jan Łukasiewicz. Both notations do not require any parentheses to define the precedence of binary operators in any possible expression. In the PN the operators precede the arguments and in the RPN they follow the arguments. The next slide presents several expressions in the traditional (infix) notation and in the corresponding RPN form.

Stack — Applications

Evaluating RPN Expressions

$2 + 2 \Rightarrow 2\ 2\ +$

$(5 - 2) * (4 + 1) \Rightarrow 5\ 2\ -\ 4\ 1\ +\ *$

$(3 + 2) * 7 \Rightarrow 3\ 2\ +\ 7\ *$

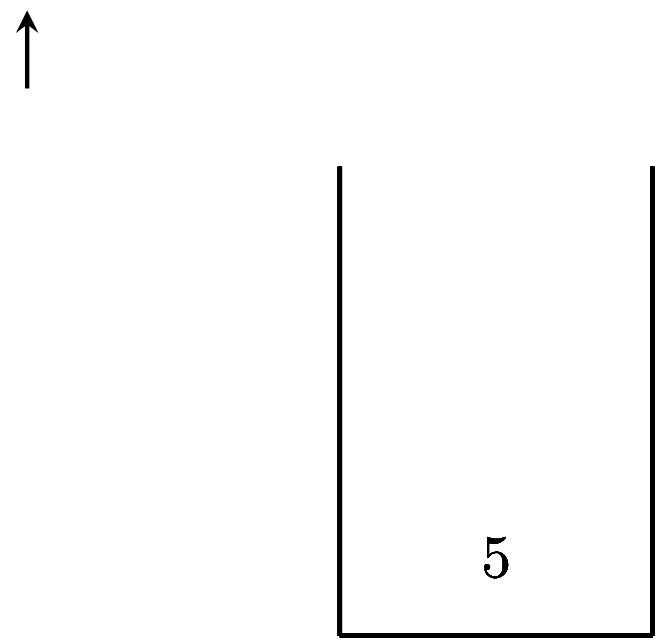
$3 + 2 * 7 \Rightarrow 2\ 7\ *\ 3\ +$

Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

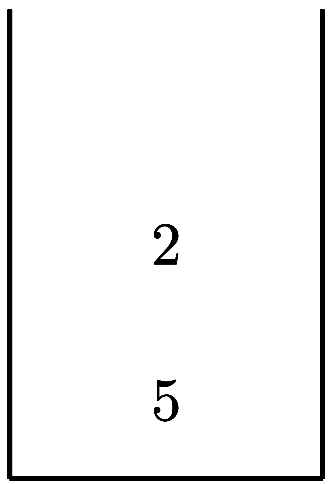


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

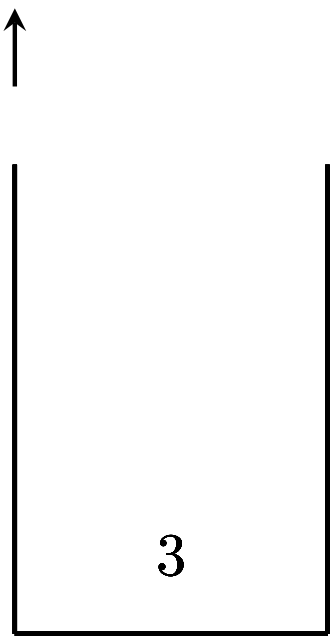


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

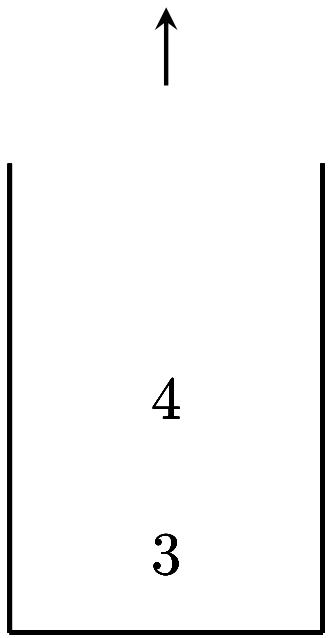


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

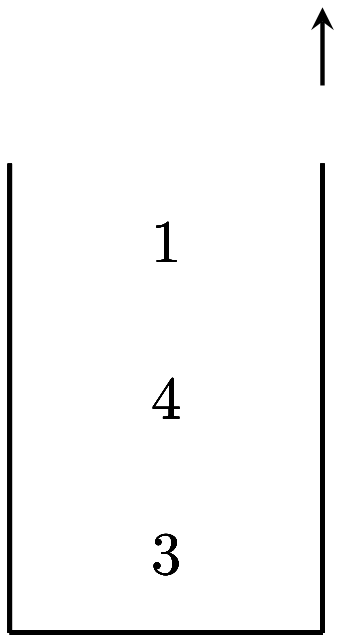


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

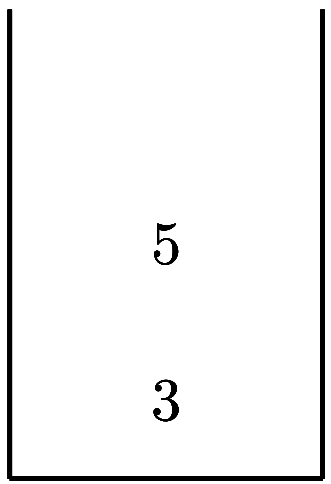


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

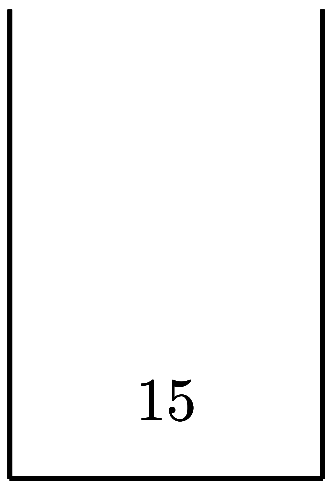


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =

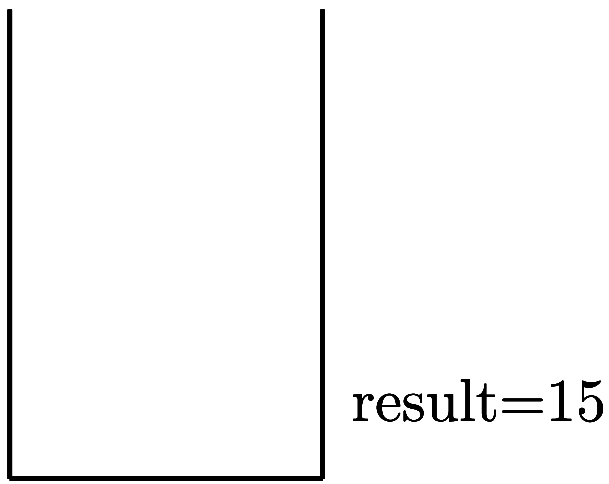


Stack — Applications

Evaluating RPN Expressions

The animation shows how an RPN expression can be evaluated with the use of the stack.

5 2 - 4 1 + * =



Stack — Applications

Evaluating RPN Expressions

As it can be observed watching the animation from the previous slide, the RPN expressions are read from the left to the right. If a token is found that is a number, then it is added to the stack, but if it is an operator, then its arguments are removed from the stack (if the RPN expression is correct, then a proper number of arguments is already stored on the stack), the operation is carried out and its result is stored back on the stack. The program presented in this lecture evaluates RPN expressions meeting the following conditions:

1. the RPN expressions consist of non-negative floating-point numbers and four operators: addition, multiplication, division and subtraction,
2. the tokens in the RPN expression are separated by single spaces,
3. the = token terminates every RPN expression and informs the program to get expression's value from the stack,
4. the program assumes that the RPN expression is correct.

Stack — Applications

Evaluating RPN Expressions

The program that evaluates RPN expressions uses only two of the stack operations, *push* and *pop*. Its code begins with three preprocessor directives, that include the `stdio.h`, `stdlib.h` and `string.h` header files. They are followed by the definition of the stack node type and `push()` and `pop()` functions.

Stack — Applications

Evaluating RPN Expressions

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct stack_node {
    double number;
    struct stack_node *next;
};
```

Stack — Applications

Evaluating RPN Expressions

```
struct stack_node *push(struct stack_node *top, double number)
{
    struct stack_node *new_node = (struct stack_node *)
        malloc(sizeof(struct stack_node));
    if (new_node) {
        new_node->number = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stack — Applications

Evaluating RPN Expressions

```
double pop(struct stack_node **top)
{
    double result = 0.0;
    if (*top) {
        result = (*top)->number;
        struct stack_node *temporary = (*top)->next;
        free(*top);
        *top = temporary;
    }
    return result;
}
```

Stack — Applications

Evaluating RPN Expressions

```
double evaluate(char *token)
{
    static struct stack_node *top = 0;
    double first_argument, second_argument;
    switch (token[0]) {
    case '+':
        top = push(top, pop(&top) + pop(&top));
        break;
    case '*':
        top = push(top, pop(&top) * pop(&top));
        break;
    case '-':
        second_argument = pop(&top);
        first_argument = pop(&top);
        top = push(top, first_argument - second_argument);
        break;
```

Stack — Applications

Evaluating RPN Expressions

```
        case '/':
            second_argument = pop(&top);
            first_argument = pop(&top);
            top = push(top, first_argument/second_argument);
            break;
        case '=':
            return pop(&top);
        default:
            top = push(top, atof(token));
        }
        return 0.0;
}
```

Stack — Applications

Evaluating RPN Expressions

The `evaluate()` function takes only one argument, which is a string representing one token of the RPN expression. It also uses a stack, whose stack pointer is a local variable called `top` and declared with the use of the `static` keyword. This keyword assures that the address stored in that pointer doesn't disappear between invocations of the `evaluate()` function. The function takes the first character in the string representing the token, and then uses the `switch` statement to recognize what it is. If it is one of the binary operators (plus, minus, times or divide), it takes the arguments from the stack, using the `pop()` function, performs the operation and stores the result on the stack, using the `push()` function. Please notice, that in case of the division and subtraction the arguments of the operations are first assigned to two local variables. This is necessary because both operations are anti-commutative, and their arguments on the stack are stored in reversed order.

Stack — Applications

Evaluating RPN Expressions

If the token is the `=` symbol, then the function removes the only node from the stack using the `pop()` function and returns the number received from this latter function, because it is the result of the RPN expression evaluation. If the token is neither of the described symbols, then it represents a floating-point number. Thus, the `evaluate()` function converts it to a number using `atof()` and stores it on the stack.

The `evaluate()` function works correctly only if it is given a valid RPN expression to evaluate. The initial value of the stack pointer in this function is `0` which is equivalent to `NULL`. The function returns 0 if it is given a token that is not the `=` symbol.

Stack — Applications

Evaluating RPN Expressions

```
double parse(char expression[])
{
    double result = 0.0;
    char *token = strtok(expression, " ");
    while (token) {
        result = evaluate(token);
        token = strtok(0, " ");
    }
    return result;
}
```

Stack — Applications

Evaluating RPN Expressions

The `parse()` functions is responsible for splitting the string, representing an RPN expression, that is passed by its parameter into several substrings representing the tokens of the expression. The delimiter character in this case is a single space. Each of the tokens is passed then to the `evaluate()` function, in the `while` loop, for recognizing. The outcome is stored in the `result` variable. When the loop stops the value of this variable is returned, because it is the value of the RPN expression.
The operation performed together by the `parse()` and `evaluate()` functions is called, in the Computer Science terminology, *parsing*.

Stack — Applications

Evaluating RPN Expressions

```c
int main(void)
{
    char rpn_expression[201];

    puts("Please enter the RPN expression:");
    scanf("%200[^\n]s", rpn_expression);
    double result = parse(rpn_expression);
    printf("Result: %.3lf\n", result);
    return 0;
}
```

Stack — Applications

Evaluating RPN Expressions

In the `main()` function the program asks the user to enter an RPN expression and then stores the string that represents it in the local variable named `rpn_expression`. Then this string is passed to the `parse()` function, whose result is stored in the `result` variable and displayed on the screen.

Queue

Like the stack, the queue (also referred to as a FIFO queue, a FIFO data structure or simply a FIFO) can be implemented in a from of a dynamic data structure. In such a case it is a list of linked nodes, where a new node is added at the end of the list (called a *tail* or a *rear*) and the operation of removing a node is performed at the beginning of the list (called a *head* or a *front*).
The implementation of a queue in the form of a dynamic data structure is explained using a program that stores in this data structure the command-line arguments.

Queue

```c
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>

#define LENGTH 500

struct fifo_node {
    char data[LENGTH];
    struct fifo_node *next;
};

struct fifo_pointers {
    struct fifo_node *head, *tail;
} fifo;
```

Queue

In the program are included four header files. The `stdio.h` is added because of the `pritnf()` function which is used to display messages on the screen. The functions necessary to dynamically allocate and deallocate memory are declared in the `stdlib.h`. Some of the functions in the program return a value of the `bool` type. That is why the `stdbool.h` is included in the program. The program also uses a function that operates on strings, so the `string.h` is added too.
The maximum length of the string (including the `'\0'` character) that can be stored in a single node of the queue is defined by the `LENGTH` constants. Its value is 500.
The type of the single node of the queue is defined as a structure with two members (lines 8–11). One of them is an array of characters (line no. 9) and the other is a pointer of the same type as the structure (line no. 10). Please notice, the similarity between the types of the single node for the stack and the queue. The pointer field has the same purpose as in the stack — it is used to link the nodes of the queue.

Queue

In order to make the implementation of the queue operations efficient two pointers are needed: one for the head and one for the tail. They can be declared separately, but it is more convenient to make them members of a structure. In the program the structure is called `fifo` and it is declared in the line no. 15, in the previous slide. The type of the structure is defined in the lines 13–15 and it has two fields that are pointers of the `struct fifo_node *` type.

Queue

```
1  void copy_string(char *destination, char *source)
2  {
3      strncpy(destination,source,LENGTH-1);
4      destination[LENGTH-1] = '\0';
5  }
```

Queue

The `copy_string()` function makes a copy of a `source` string in the `destination` array of character in a safer way than the `strcpy()` function does. It copies at most `LENGTH-1` characters from the `source` string and then terminates the string by adding the `'\0'` character in the `destination` array, in case the `source` string wasn't terminated in such a way.
The function may truncate the original string. In case of the program, that is described here, it is not an issue, but in other kind of software it may be, so the function has to be applied carefully in other programs.

Queue

The Enqueue Operation

The operation of adding a new node to the queue is called *enqueue*. Its implementation has to satisfy the following assertions:

- If the queue has at least one node, the operation adds a new element at the back of it, and if the queue is empty the function creates and adds its first node.
- If the implementation fails to create a new node, then the queue stays the same as it was.
- If the operation of adding a new node is successful then the queue length increases by one element.

Queue

The Enqueue Operation

```
1  bool enqueue(struct fifo_pointers *fifo, char *data)
2  {
3      struct fifo_node *new_node = (struct fifo_node *)
4          malloc(sizeof(struct fifo_node));
5      if(new_node) {
6          copy_string(new_node->data,data);
7          new_node->next = NULL;
8          if(fifo->head == NULL && fifo->tail == NULL) {
9              fifo->head = fifo->tail = new_node;
10         } else {
11             fifo->tail->next = new_node;
12             fifo->tail = new_node;
13         }
14         return true;
15     } else
16         return false;
17 }
```

Queue

The Enqueue Operation

The enqueue operation is implemented as a function of the same name. The function takes two arguments that are passed by its parameters. The first one is the address of the queue pointer structure. The second one is a pointer to the string that should be stored in the new node of the queue. The return type of the `enqueue()` function is `bool`.

The function first tries to allocate memory for the new node of the queue (lines no. 3 and 4). In the 5^{th} line it checks if the operation has been successful. If not it returns `false` (line no. 16) and exits. Otherwise, the function calls `copy_string()` to store the string passed by the `data` parameter into the `data` field of the new node (line no. 6) and initializes the `next` field with `NULL`. Then it needs to recognize which of the two possible cases it should handle:

1. the node ought to be added at the end of the queue,
2. the node ought to be added to an empty queue.

Queue

The Enqueue Operation

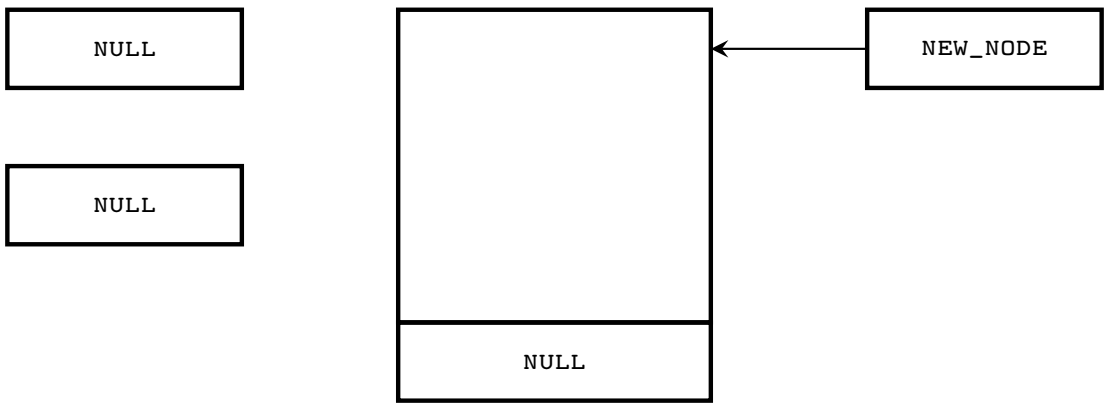
To do so it checks the value of the queue pointers (line no. 8). If the `head` and `tail` pointers are empty, than the first case should be handled, and the `enqueue()` function assigns to both of them the address of the new node (line no. 9). This operation is also illustrated in the next slide.

In the second case (lines 11-12) the `enqueue()` function first stores the address of the new node in `next` member of the last node (line no. 11). This way it appends the new node to the queue. However, now the `tail` pointer points to a wrong node — it should always point to to last node in the queue. That is why in the 12^{th} line the `enqueue()` function also assigns to the pointer the address of the new node. This operation is shown in the second next slide.

Please notice, that in both cases the `next` field of the new node has to be set to `NULL`, because this value indicates that it is the last node in the queue. Also in both cases the function returns `true` after successfully adding a new node to the queue.

Queue

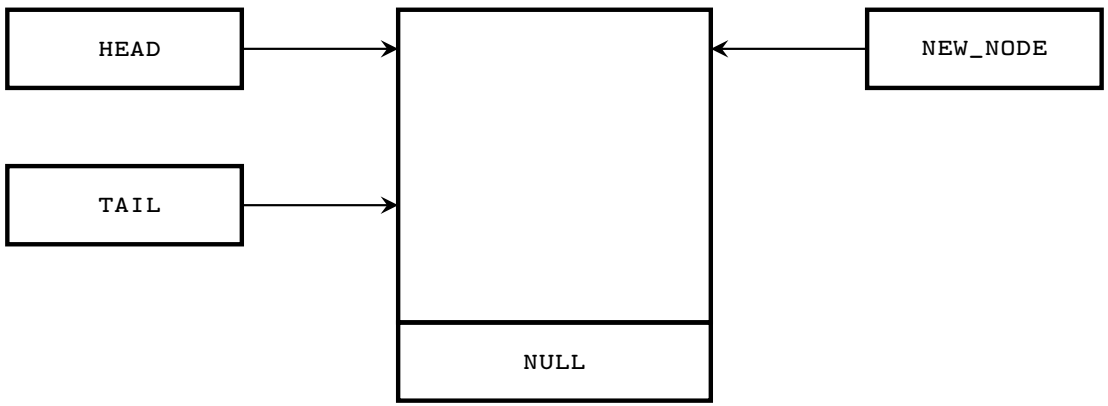
The `enqueue()` Function — Adding Node To The Empty Queue



The queue before the line no. 9 of the `enqueue()` function is performed

Queue

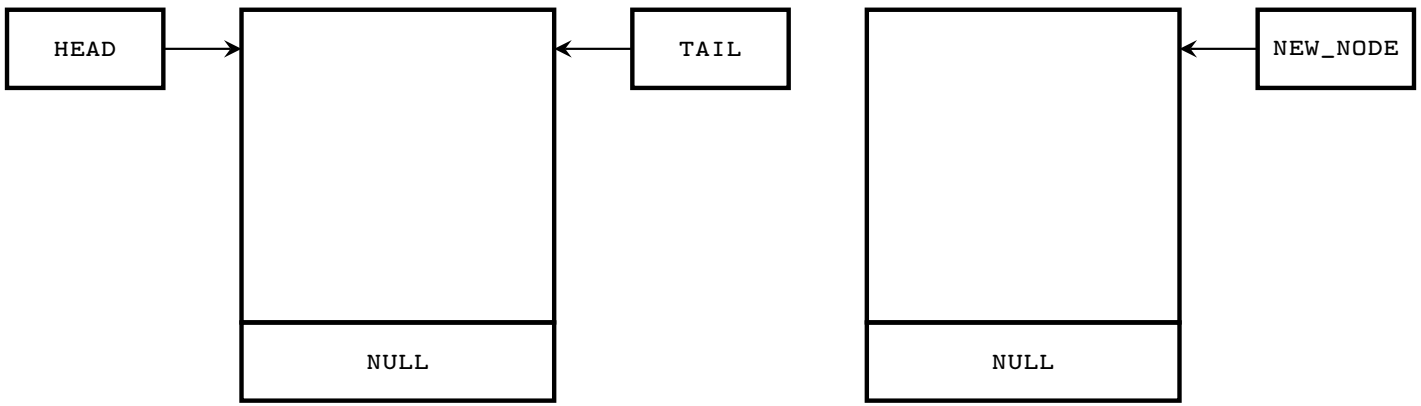
The `enqueue()` Function — Adding Node To The Empty Queue



The queue after the line no. 9 of the `enqueue()` function is performed

Queue

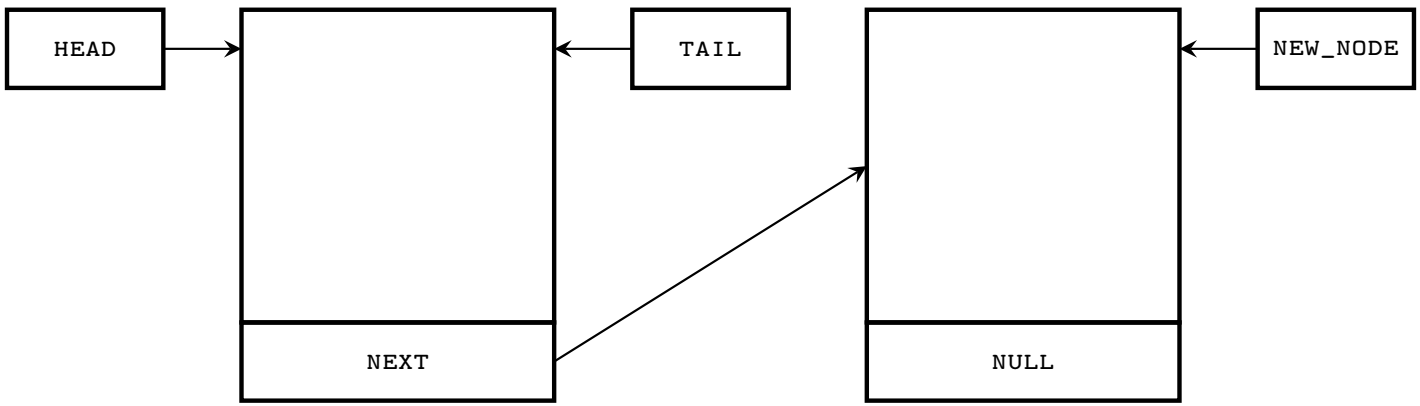
The `enqueue()` Function — Appending The New Element



Before the line no. 11 of the `enqueue()` function is performed

Queue

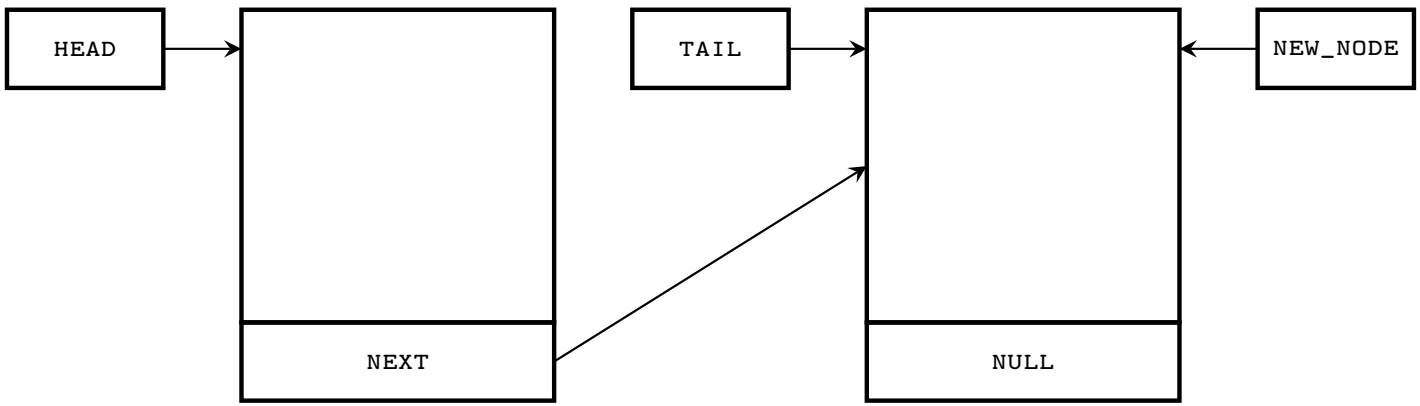
The `enqueue()` Function — Appending The New Element



After the line no. 11 of the `enqueue()` function is performed

Queue

The `enqueue()` Function — Appending The New Element



After the line no. 12 of the `enqueue()` function is performed

Queue

The Dequeue Operation

The operation of removing a node from the queue is called *dequeue*. It should meet the following assertions:

- If the queue is empty then the state of its pointers should not change after the operation is performed — both pointers have to have the value of `NULL`.
- If a node is removed from a queue that has only one element, then both queue pointers must be set to `NULL`.
- If a node is removed from a queue that has more than one element, than the queue length is reduced by one node and the pointers correctly point to the head and tail of the queue.

Queue

The Dequeue Operation

```
1  bool dequeue(struct fifo_pointers *fifo, char *data)
2  {
3      if(fifo->head) {
4          struct fifo_node *temporary = fifo->head->next;
5          copy_string(data,fifo->head->data);
6          free(fifo->head);
7          fifo->head = temporary;
8          if(temporary == NULL)
9              fifo->tail = NULL;
10         return true;
11     }
12     return false;
13 }
```

Queue

The Dequeue Operation

The dequeue operation is implemented as a function of the same name. It takes two arguments. The first one is the address of the queue pointers structure, that is passed by its first parameter. The second one is the address of a character array, passed by its second parameter.

First the function checks if the `head` pointer, that should point to the first node in the queue, is not empty (line no. 3). If so, it assigns the address stored in the `next` field of the first node to a local pointer named `temporary` (line no 4). This is the address of the second node in the queue. Then the `dequeue()` function copies the string from the `data` member of the first node, to the array whose address is passed by its second parameter (line no. 5). Next the function deallocates the memory allocated for the queue first node (line no. 6). After that operation is performed, the `head` pointer is invalid, because the first node no longer exists.

Queue

The Dequeue Operation

The address of the non-existing node in the `head` pointer has to be replaced with the address stored in the `temporary` pointer (line no. 7). It is the address of previously the second node of the queue and now the first node. However, it may happen that the `enqueue()` function has removed the only node in the queue, which becomes empty. That means that after the 7^{th} line is performed, both the `head` and `temporary` pointers are empty, but the `tail` pointer is invalid. It points to a non-existent node. To find out if that is the case, the `dequeue()` checks the `temporary` pointer (line no. 8), and if it is empty, the function sets the `tail` pointer to `NULL`. Finally, the `dequeue()` returns `true` informing that it successfully removed a node from the queue. If the condition in the line no. 3 is not met, then it means that the queue is empty and the function returns `false` (line no. 12) because it is unable to remove a node from an empty queue.

Queue

```
1  int main(int argc, char *argv[])
2  {
3      for(int i=0; i<argc; i++)
4          if(!enqueue(&fifo, argv[i]))
5              printf("Error adding the argument: %s to the queue!",
6              argv[i]);
7      while(fifo.head) {
8          char string[LENGTH];
9          if(dequeue(&fifo,string))
10             printf("Data from queue: %s\n", string);
11     }
12     return 0;
13 }
```

Queue

The `main()` function adds in the `for` loop command-line arguments to the queue (lines 3–6), verifying in each iteration that the operation has been successfully completed (line no. 4). Then it retrieves them from the queue in the `while` loop (lines 7–11) and displays on the screen. Please notice, that the path to the executable file of the program is also added to the queue — it is pointed by the `argv[0]`. The `while` loop is performed as long as the `head` pointer is not empty. Additionally, the function checks in the loop's body if the `dequeue()` returned `true`, before printing the string stored in the `string` array.

Summary

Other implementations of the described data structures are possible. Both the stack and the queue may be implemented in hardware or software. It is also possible to build them on top of a statically or dynamically allocated array. In that case the nodes of the stack or queue are elements of the array, and instead of pointers the indices are used.

Both data structures have many applications, in compilers, operating systems and other software. There is also an algorithm, invented by Edsger Dijkstra that allows the program to convert an infix expression to postfix notation. It is called the *shunting-yard algorithm* and it also uses a stack. Unlike the algorithm for evaluating the postfix expressions, the shunting-yard algorithm stores in this data structure operators and parentheses instead of numbers.

Questions

?

The End

Thank You For Your Attention!