Урок 5

Динамические структуры данных

Стек, очередь, дек

Стек — это структура данных, которая работает по принципу «первый пришёл — последний ушел» (First In — Last Out). Стек удобно использовать в некоторых алгоритмах, поэтому полезно уметь создавать эту структуру. Рассмотрим создание стека на основе массива.

Типовые операции:

1. Push — добавление в стек.
2. Pop — изъять из стека.
3. Проверка наличия элементов.
4. Очистка.

Очередь — структура данных, также организованная по принципу FIFO (First In — First Out). Это линейный список, для которого введены две операции:

1. Добавление нового элемента в конец очереди.
2. Удаление первого элемента из очереди.

Операционные системы используют очереди для организации сообщений между программами: каждая программа имеет свою очередь сообщений. Контроллеры жестких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создается очередь из пакетов данных, ожидающих отправки.

Типовые операции:

1. Enqueue — добавление в очередь.
2. Dequeue — изъятие из очереди.
3. Проверка наличия элементов.
4. Очистка.

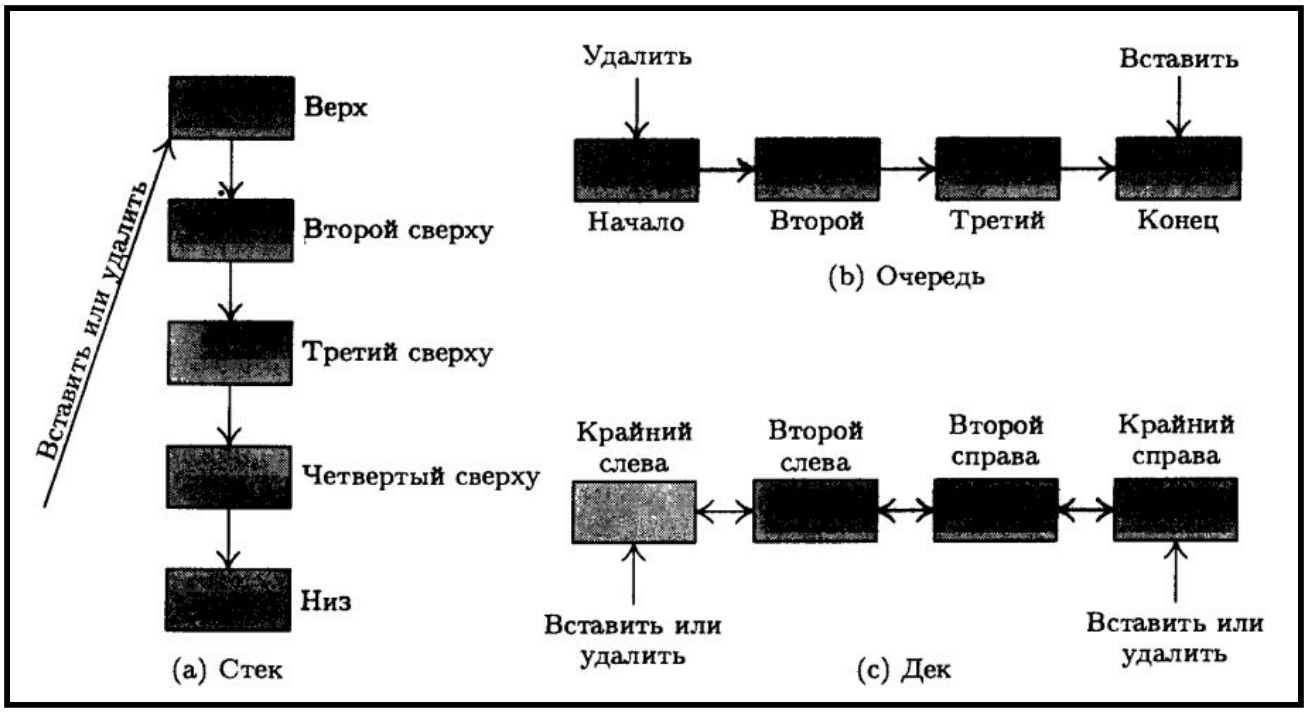
Дек (двухсторонняя очередь) — линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт.

Типовые операции:

1. pushBack — добавление в конец очереди.
2. pushFront — добавление в начало очереди.
3. popBack — выборка с конца очереди.
4. popFront — выборка с начала очереди.
5. Проверка наличия элементов.
6. Очистка.

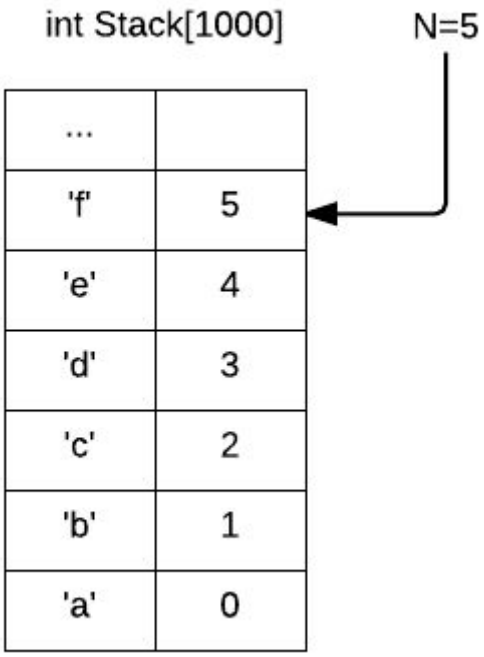
Три наиболее важных класса линейных списков:



Примеры реализаций структур данных без использования динамических структур данных

Создание стека с использованием массива

В примере создадим одномерный массив из 1000 элементов (максимальный размер стека). Глобальная переменная N является одновременно и количеством элементов, и «указателем» на последний элемент в массиве. Указателем в переносном смысле: просто с помощью N можно узнать, какой элемент лежит на вершине стека.



Обратите внимание на строчку:

```
#define T char
```

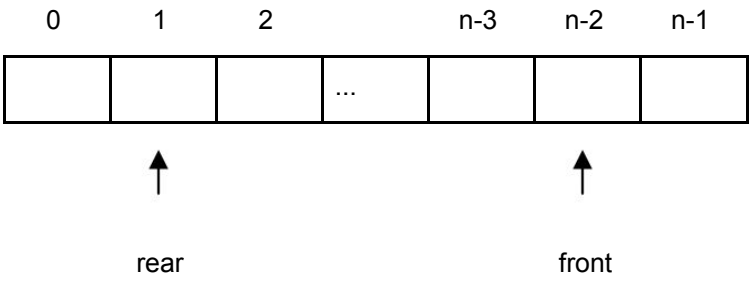
Мы определяем некоторый тип в качестве T и пишем программу, используя его как тип стека. Если нам нужно будет изменить тип стека, достаточно изменить его тип в директиве компилятора.

```c
#include <stdio.h>

#define T char
#define MaxN 1000

T Stack[MaxN];

int N = -1;

void push(T i)
{
    if (N < MaxN)
    {
        N++;
        Stack[N] = i;
    }
    else
        printf("Stack overflow");
}

T pop()
{
    if (N != -1)
        return Stack[N--];
    else
        printf("Stack is empty");
}

int main(int argc, char *argv[])
{
    T c;
    push('a');
    push('b');
    push('c');
    push('d');
    push('e');
    push('f');
    while(N != -1)
        printf("%c",pop());
    return 0;
}
```

Средства языка C позволяют создать структуру Stack (struct Stack), которая хранит как сами элементы, так и их количество. Благодаря этому можно создавать любое количество стеков.

Реализация очереди через массив

В этой реализации мы создаем массив q[n], который будет определять размер очереди.



Переменная front указывает на голову очереди, а rear — на элемент, который заполнится (конец очереди), когда в очередь добавляется новый элемент. При добавлении нового элемента он записывается в очередь в q[rear], а rear увеличивается на единицу. Если значение rear становится больше n, то мы как бы циклически обходим массив, и значение переменной становится равным 0. Извлечение элемента из очереди производится аналогично: после извлечения элемента q[front] из очереди переменная front увеличивается на 1. Если значение front становится больше n, то мы как бы циклически обходим массив, и значение переменной становится равным 0.

Реализация двухсторонней очереди

Для реализации двухсторонней очереди можно прибегнуть к разным алгоритмам:

1. Реализовать за счёт использования структуры стек, в котором можно добавлять и удалять элементы с обоих концов. Правда, в этом случае лучше использовать динамические структуры данных (см. далее).
2. Реализовать за счёт использования очереди, в которой также можно добавлять и удалять данные с обоих концов.
3. Реализовать за счет статического массива, данные в котором будут сдвигаться при добавлении элементов.
4. Использовать динамический двусвязный список (см. далее)

Динамические структуры данных

Довольно часто нужно решать задачи, в которых заранее неизвестно, сколько элементов данных будет использовано. Например, мы создавали с вами стек большего размера, чем он мог реально понадобиться.

Частично мы можем решить эту проблему с помощью динамических массивов, но и в этом случае нам нужно заранее знать, сколько элементов будет в массиве. Можно создавать каждый раз новый массив, увеличивая или уменьшая его размер, но тогда нужно будет копировать все элементы из старого массива в новый.

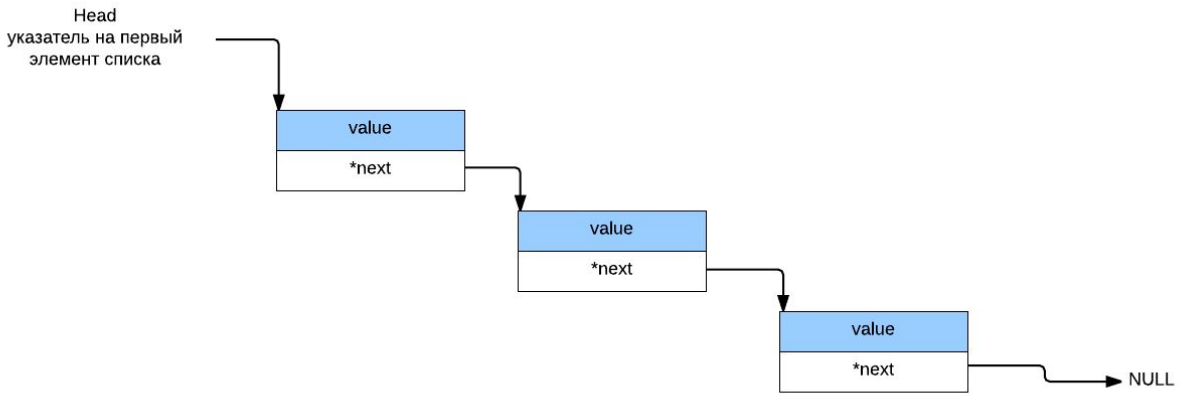
Когда количество элементов заранее неизвестно, используют динамические структуры данных.

Списки

Односвязный список

В односвязном списке каждый элемент информации содержит ссылку на следующий элемент списка. Каждый элемент данных обычно представляет собой структуру, которая состоит из информационных полей и указателя связи.

Односвязный список из трёх элементов:

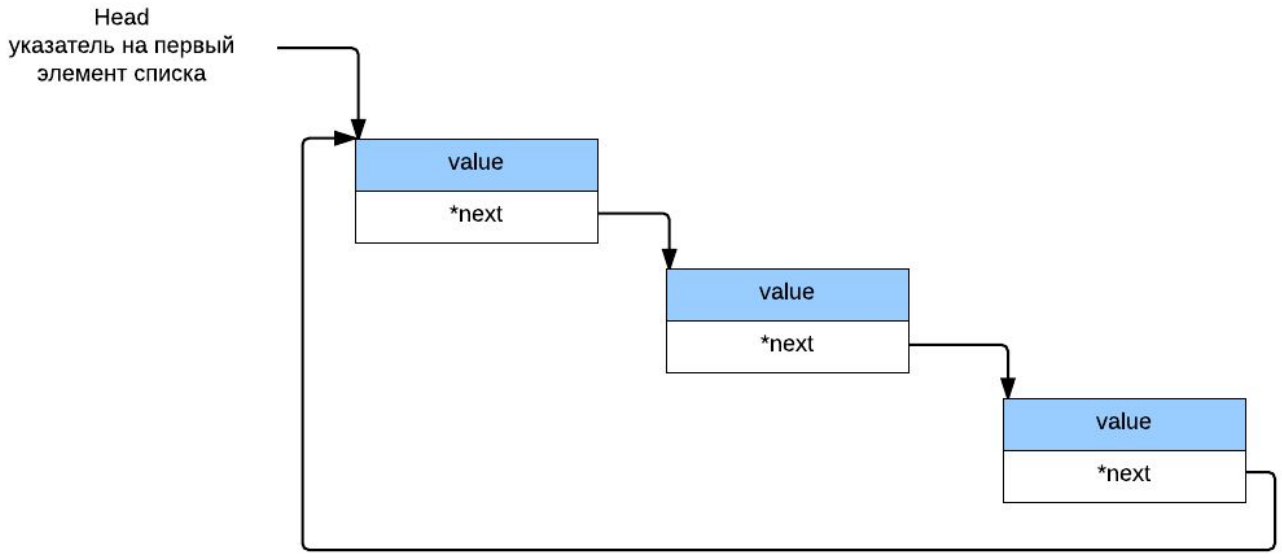


В односвязном списке используется структура, которая содержит данные и переменную-указатель.

```
struct TNode
{
    int value;          // Данные
    struct TNode* next; // Указатель на следующий элемент списка
};
struct TNode* Head=NULL;
```

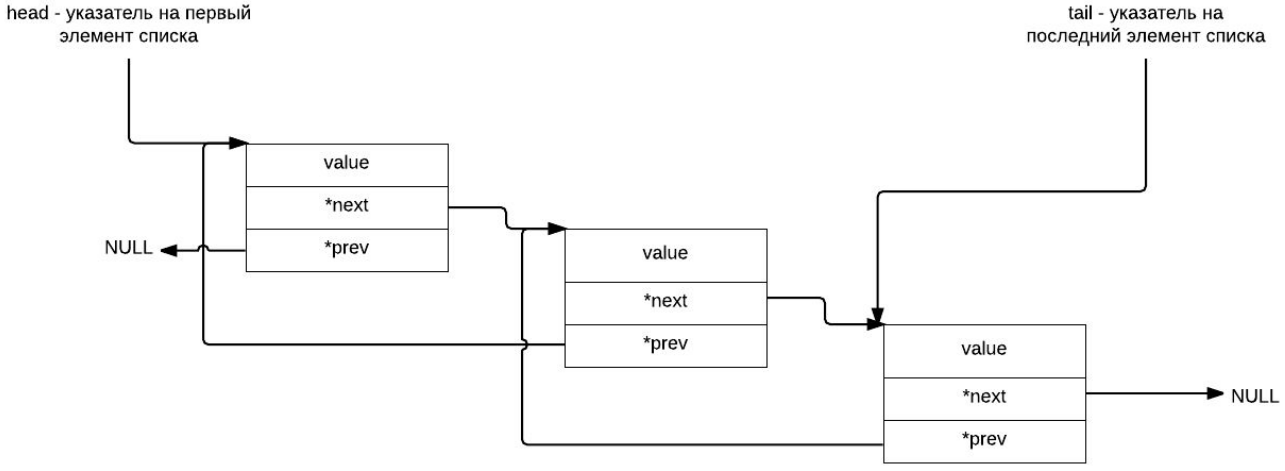
Циклический список

Если замкнуть связный список в кольцо так, чтобы последний элемент содержал ссылку на первый, то получается циклический список.



Двусвязный список

Поскольку элементы односвязного списка содержат ссылки только на следующий элемент, к предыдущему перейти нельзя. Поэтому перебор возможен только в одном направлении. Этот недостаток устранён в двусвязном списке, где каждый элемент хранит адрес как следующего, так и предыдущего элемента.



Для такого списка обычно хранят два адреса: «голова» списка (указатель head) и его «хвост» (указатель tail).

```
struct TNode
{
    int value;          // Данные
    struct TNode *next; // Указатель на следующий элемент списка
    struct TNode *prev; // Указатель на предыдущий элемент списка
};
struct TNode *head;
struct TNode *tail;
```

Рассмотрим пример создания стека на основе динамической структуры «односвязный список».

Стек на основе односвязного списка

Создадим стек на основе односвязного списка.

Напомним, что стек представляет собой специально организованный по принципу First In — Last Out («первым пришёл — последним ушёл») вид памяти. Для организации стека достаточно создать односвязный список и реализовать две команды: добавления на вершину списка (Push) и извлечения с вершины списка (Pop). Так как в односвязном списке мы должны хранить его вершину, мы легко реализуем эти две команды.

Для работы с динамическими структурами данных создадим узел этих данных:

```
struct TNode
{
    int value;          // Данные
    struct TNode *next; // Указатель на следующий элемент списка
};
```

Самым трудным для понимания здесь является то, что внутри структуры Node мы создаём указатель на самих себя.

Чтобы не писать в программе struct TNode, дадим этой структуре имя Node:

```
typedef struct TNode Node;
```

Структура Stack

Опишем структуру Stack и глобальную переменную Stack. Это позволит структурировать программу, а также, при желании, даст возможность создавать несколько независимых стеков.

```
struct Stack
{
    Node *head;
    int size;
    int maxSize;
};
struct Stack Stack;
```

В главной программе в поля структуры Stack запишем максимальный размер стека и инициализируем head пустой ссылкой:

```
Stack.maxSize = 100;
Stack.head = NULL;
```

Добавление элемента. Push

Проверяем, что мы можем добавлять элемент в стек. В противном случае выводим сообщение об ошибке и завершаем выполнение функции.

```
if (Stack.size >= Stack.maxSize) {
    printf("Error stack size");
    return;
}
```

Создаём указатель на узел:

```
Node *tmp;
```

Выделяем для него место в памяти и сохраняем ссылку:

```
Tmp = (Node*) malloc(sizeof(Node));
```

Записываем данные в новый узел:

```
tmp->value = value;     // записываем данные
tmp->next = Stack.head; // записываем предыдущее значение главы
```

head теперь указывает на вновь созданный элемент:

```
Stack.head = tmp; // голова теперь указывает на вновь созданный элемент
Stack.size++;     // увеличиваем количество элементов в стеке
```

Извлечение элемента. Pop

Проверяем размер стека. Если он пуст, то выводим соответствующее сообщение и ничего не делаем.

```
if (Stack.size == 0)
{
    printf("Stack is empty");
    return;
}
```

Создаём временный указатель:

```
Node* next = NULL;
```

Так как мы должны удалить вершину списка, требуется предварительно сохранить данные из вершины:

```
T value;
Value = Stack.head->value;
```

Запоминаем значение головы во временной переменной:

```
Next = Stack.head;
```

Записываем в голову предыдущий элемент:

```
Stack.head = Stack.head->next;
```

А запись, на которую указывала голова, удаляем, освобождая память:

```
free(next);
```

Возвращаем значение, которое было в голове, и уменьшаем количество элементов:

```
Stack.size--;
return value;
```

Распечатка односвязного списка

Для распечатки стека нам нужно пробежаться по всем элементам списка, начиная с головы, до тех пор, пока мы не найдём элемента, значение поля next у которого будет равно NULL.

```
void PrintStack(struct Stack Stack)
{
    Node *current = Stack.head;
    while(current != NULL)
    {
        printf("%c ", current->value);
        current = current->next;
    }
}
```

Полный текст программы:

```
#include <stdio.h>
#include <malloc.h>
#define T char
// Опишем структуру узла списка
struct TNode
{
    T value;
    struct TNode *next;
};
typedef struct TNode Node;

struct Stack
{
    Node *head;
    int size;
    int maxSize;
};
struct Stack Stack;

void push(T value)
```

```
{
    if (Stack.size >= Stack.maxSize) {
        printf("Error stack size");
        return;
    }
    Node *tmp = (Node*) malloc(sizeof(Node));
    tmp->value = value;
    tmp->next = Stack.head;
    Stack.head = tmp;
    Stack.size++;
}

T pop() {
    if (Stack.size == 0)
    {
        printf("Stack is empty");
        return;
    }
    // Временный указатель
    Node* next = NULL;
    // Значение "наверху" списка
    T value;
    Value = Stack.head->value;
    Next = Stack.head;
    Stack.head = Stack.head->next;
    // Запись, на которую указывала голова удаляем, освобождая память
    free(next);
    // Возвращаем значение, которое было в голове
    Stack.size--;
    return value;
}

void PrintStack()
{
    Node *current = Stack.head;
    while(current != NULL)
    {
        printf("%c ", current->value);
        current = current->next;
    }
}

int main(int argc, char *argv[])
{
    Stack.maxSize = 100;
    Stack.head = NULL;
    push('a');
    push('b');
    push('c');
    push('d');
    push('e');
    push('f');
    PrintStack();
    return 0;
}
```

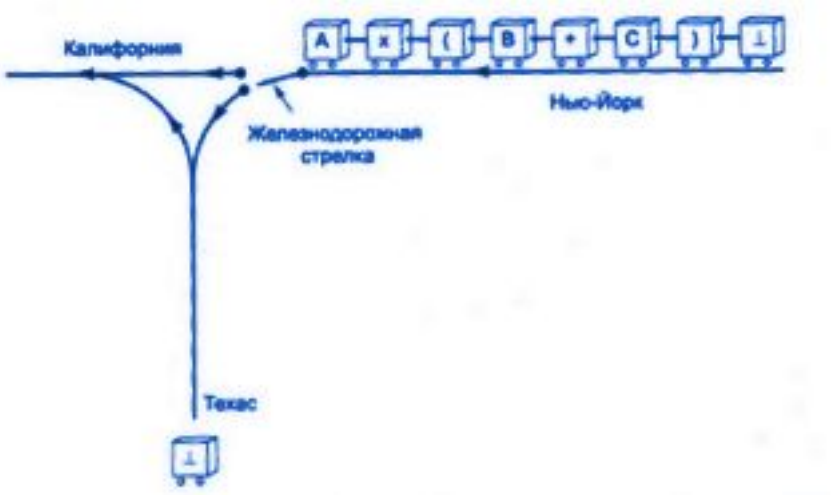
Перевод из инфиксной записи в постфиксную

Постановка задачи

На вход программы поступает выражение, состоящее из односимвольных идентификаторов и знаков арифметических действий. Требуется преобразовать это выражение в обратную польскую запись или же сообщить об ошибке.

Алгоритм перевода в обратную польскую запись

Существует несколько алгоритмов для превращения инфиксных формул в обратную польскую запись. Мы рассмотрим переработанный алгоритм, идея которого предложена Эдсгером Дейкстра (E.W. Dijkstra). Предположим, что формула состоит из переменных, двухоперандных операторов +, -, *, /, ^, а также левой и правой скобок. Чтобы отметить конец формулы, мы будем вставлять символ ⊥ после её последнего символа и перед первым символом следующей формулы.



На рисунке схематично показана железная дорога из Нью-Йорка в Калифорнию с ответвлением, ведущим в Техас. Каждый символ формулы представлен одним вагоном. Поезд движется на запад (налево). Перед стрелкой каждый вагон должен останавливаться и узнавать, должен ли он двигаться прямо в Калифорнию, или ему нужно будет по пути заехать в Техас. Вагоны, содержащие переменные, всегда направляются в Калифорнию и никогда не едут в Техас. Вагоны, содержащие все прочие символы, должны перед прохождением стрелки узнавать о содержимом ближайшего вагона, отправившегося в Техас.

В таблице показана зависимость ситуации от того, какой вагон отправился в Техас последним, и какой вагон находится у стрелки. Первый вагон (помеченный символом ⊥) всегда отправляется в Техас.

стрелка> Техас \/	⊥	+	P	*	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
P	2	2	2	1	1	1	2
*	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Числа соответствуют следующим ситуациям:

1. Вагон на стрелке отправляется в Техас.
2. Последний вагон, направившийся в Техас, разворачивается и направляется в Калифорнию. Остаёмся на том же символе на стрелке.
3. Вагон, находящийся на стрелке, и последний вагон, отправившийся в Техас, угоняются и исчезают.
4. Остановка. Символы, находящиеся на калифорнийской ветке, представляют собой формулу в обратной польской записи, если читать слева направо.
5. Остановка. Произошла ошибка. Изначальная формула была некорректно сбалансирована.

После каждого действия проводится новое сравнение вагона, находящегося у стрелки (это может быть тот же вагон, что и в предыдущем сравнении, или следующий), и вагона, который на данный момент последним ушёл на Техас. Этот процесс продолжается до тех пор, пока не будет достигнут шаг 4. Отметим, что линия на Техас используется как стек, где отправка вагона в Техас — это помещение элемента в стек, а разворот отправленного в Техас вагона в сторону Калифорнии — это выталкивание элемента из стека.

Порядок следования переменных в инфиксной и постфиксной записи одинаков. Однако порядок следования операторов не всегда один и тот же. В обратной польской записи операторы появляются в том порядке, в котором они будут выполняться.

Предлагается реализовать перевод в постфиксную запись самостоятельно в качестве домашней работы.

Примеры использования структур данных

Пример создания структуры Stack на основе массива с использованием структуры

Если нам нужно в программе использовать несколько стеков, то мы можем оформить структуру Stack и в дальнейшем создавать переменные данной структуры.

```
#include <stdio.h>
#define MaxN 1000
#define T char

struct TStack
{
    int N;                      // Номер верхнего элемента
    T Data[MaxN];               // Данные
};
struct TStack Stack1;
struct TStack Stack2;
struct TStack Stack3;

void push(struct TStack *Stack, T data)
{
    Stack->N++;
    (*Stack).Data[(*Stack).N] = data;
    // Или проще
    // Stack->Data[Stack->N] = data;
}
T pop(struct TStack *Stack)
{
    if (Stack->N != -1)
        return Stack->Data[Stack->N--];
}
void init(struct TStack *Stack)
{
    Stack.N = -1;
}
int main(int argc, char *argv[])
{
    init(&Stack1);
    init(&Stack2);
    init(&Stack3);
    push(&Stack1, 'a');
    push(&Stack1, 'b');
    push(&Stack2, 'c');
    push(&Stack3, 'd');
    push(&Stack1, 'e');
    push(&Stack1, 'f');
    while(Stack1.N != -1)
        printf("%c", pop(&Stack1));
    return 0;
}
```

Вычисление выражения, записанного в постфиксной записи

Дана строка выражения, записанного в постфиксной записи. Написать программу, которая вычисляет это выражение.

Пример: 2 2 + (=4)

Решение:

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <ctype.h>
#include <stdlib.h>

#define T int

struct TNode
{
    T value;                    // Данные
    struct TNode *next;         // Указатель на следующий элемент списка
};

typedef struct TNode Node;

Node* head = NULL;

void Push(T value)
{
    Node *temp;
    temp = (Node*)malloc(sizeof(Node));
    temp->value = value;
    temp->next = head;
    head = temp;
}

T Pop()
{
    Node* temp = NULL;
    T value = head->value;
    temp = head;
    head = head->next;
    free(temp);
    return value;
}

void Print()
{
    Node* current;
    current = head;
    while (current != NULL)
    {
        printf("%d ", current->value);
        current = current->next;
    }
}
```

```
void PrintR(Node* current)
{
    if (current != NULL)
    {
        PrintR(current->next);
        printf("%d ", current->value);
    }
}

int isNumber(char *str)
{
    int i = 0;
    while (str[i] != '\0')
        if (!isdigit(str[i++]))
            return 0;
    return 1;
}

int main(int argc, char *argv[])
{
    int res = 0;
    // 2+2 = 2 2 +
    // (2+2)*2=2 2 + 2 *
    char buf[100] = "20 30 - 10 *"; // (20-30)*10
    printf("Input postfix expression:");
    // Выражение разделенное пробелами(по одному пробелу!)
    // gets_s(buf);
    for (int i = 0; I < strlen(buf); i++)
    {
        // получаем элемент из строки
        char el[20];          // элемент(число или операция)
        int j = 0;
        while (buf[i] != ' ' && buf[i] != '\0')
        {
            el[j] = buf[i];
            j++;
            i++;
        }
        el[j] = '\0';
        // Если элемент Число
        if (isNumber(el))
            Push(atoi(el));   // Кладём его в стек, преобразовав из строки в
integer
        else
        {
            switch (el[0])
            {
                case '+':
                    res = Pop() + Pop();
                    Push(res);
                    break;
                case '-':
                    res = -Pop() + Pop();
                    Push(res);
                    break;
                case '*':
                    res = Pop() * Pop();
                    Push(res);
```

```
                    break;
                case '/':
                    res = Pop() / Pop();
                    Push(res);
                    break;
                default:
                    break;
            }
        }
    }
    printf("%d", Pop());
    getchar();
    return 0;
}
```

***Задача МГУ ВМК

Рассмотрим работу с двусвязным списком на примере решения реальной задачи МГУ ВМК.

Задание: написать программу, загружающую из файла последовательность символьных строк ограниченной длины и располагающую их элементы в динамически создаваемом двусвязном списке. Отсортировать список и вывести его на экран.

Мы предоставим часть решения. Сортировку списка вам предстоит сделать самостоятельно.

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>

struct TDblNode {
    char string[256];
    struct TDblNode *next;
    struct TDblNode *prev;
};
typedef struct TDblNode Node;

struct DblList
{
    Node *head, *tail, *current;
};
struct DblList List;

/*Основные действия, производимые над узлами ДС:
инициализация списка;
добавление узла в список;
удаление узла из списка;
удаление корня списка;
вывод элементов списка;
вывод элементов списка в обратном порядке;
взаимообмен двух узлов списка*/
// Инициализация ДС
Node* Init(char* a)
{
    Node *temp;
    // Выделение памяти под корень списка
    temp = (Node*) malloc(sizeof(Node));
    // Перенос строки
```

```
    strcpy(temp->string, a);
    temp->next = NULL;
    temp->prev = NULL;
    return temp;
}

/*Добавление узла в ДС
Функция добавления узла в список принимает два аргумента:
указатель на узел, после которого происходит добавление;
данные для добавляемого узла.*/
/*Добавление узла в ДС включает в себя следующие этапы:
создание узла добавляемого элемента и заполнение его поля данных;
переустановка указателя «следующий» узла, предшествующего добавленному, на добавляемый узел;
переустановка указателя «предыдущий» узла, следующего за добавляемым, на добавляемый узел;
установка указателя «следующий» добавляемого узла на следующий узел (тот, на который указывал предшествующий узел);
установка указателя «предыдущий» добавляемого узла на узел, предшествующий добавляемому (узел, переданный в функцию).*/

Node* AddElem(Node *current, char *a)
{
    Node *temp, *p;
    Temp = (Node*) malloc(sizeof(Node));
    p = current->next;
    current->next = temp;
    strcpy(temp->string, a); // Перенос строки
    temp->next = p;
    temp->prev = current;
    if (p!=NULL)
        p->prev = temp;
    current = temp;
    return current;
}

/*Удаление узла ДС включает в себя следующие этапы:
установка указателя «следующий» предыдущего узла на узел, следующий за удаляемым;
установка указателя «предыдущий» следующего узла на узел, предшествующий удаляемому;
освобождение памяти удаляемого узла.*/

// Вывод элементов ДС
void ListPrint()
{
    Node *p = List.head;
    do
    {
        if (p == List.current)
            printf("*");
        printf("%s ", p->string);
        p = p->next;
    }
    // Условие окончание обхода
    while(p!=NULL);
}

int main(void)
{
    char buf[256];
```

```
    FILE* file;
    file = fopen("D:\\temp\\str.txt", "r");
    fscanf(file, "%s", buf);
    List.head = Init(buf);
    List.current = List.head;
    List.tail = List.head;
    while (fscanf(file, "%s", buf) != EOF) {
        List.current = AddElem(List.current, buf);
    }
    ListPrint();
    printf("\n");
    return 0;
}
```

Домашняя работа

1. Реализовать перевод из десятичной в двоичную систему счисления с использованием стека.
2. Написать программу, которая определяет, является ли введенная скобочная последовательность правильной. Примеры правильных скобочных выражений: (), ([])(), {}(), ([{}]), неправильных —)(, ())({), (,])}), ([(]) для скобок [, (, {.
   Например: (2+(2*2)) или [2/{5*(4+7)}].
3. *Создать функцию, копирующую односвязный список (то есть создающую в памяти копию односвязного списка без удаления первого списка).
4. *Реализовать алгоритм перевода из инфиксной записи арифметического выражения в постфиксную.
5. Реализовать очередь:
   1. С использованием массива.
   2. *С использованием односвязного списка.
6. ***Реализовать двустороннюю очередь.

Дополнительные материалы

1. Обратная польская запись

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Род Стивенс, Алгоритмы. Теория и практическое применение. Издательство «Э», 2016.
2. Никлаус Вирт, Алгоритмы и структуры данных. Новая версия для Оберона, ДМК, Москва, 2010.
3. Пол Дейтел, Харви Дейтел. С для программистов. С введением в С11, ДМК, Москва, 2014.