**Оглавление**

[Глава 1. Исходные данные варианта задания 4](#_Toc27707231)

[Глава 2. Теоретическое введение 4](#_Toc27707232)

[2.1 Абстрактный тип данных 4](#_Toc27707233)

[2.2 Принципы анализа эффективности алгоритма 6](#_Toc27707234)

[2.3 Сортировка простой вставкой 9](#_Toc27707235)

[Глава 3. Экспериментальная часть 11](#_Toc27707236)

[3.1 Листинг программы 11](#_Toc27707237)

[3.2 Анализ сложности алгоритма 17](#_Toc27707238)

[3.3 Особые случаи 20](#_Toc27707239)

[Глава 4. Вывод 23](#_Toc27707240)

[Список литературы 24](#_Toc27707241)

[Использованные источники 25](#_Toc27707242)

# **Глава 1. Исходные данные варианта задания**

|  |  |  |  |
| --- | --- | --- | --- |
| № варианта | Реализация связи элементов линейного списка | Способ организации линейного связанный список | Алгоритм сортировки |
| 57. | Указатели | Дек | Простая вставка |

# **Глава 2. Теоретическое введение**

## **2.1** **Абстрактный тип данных**

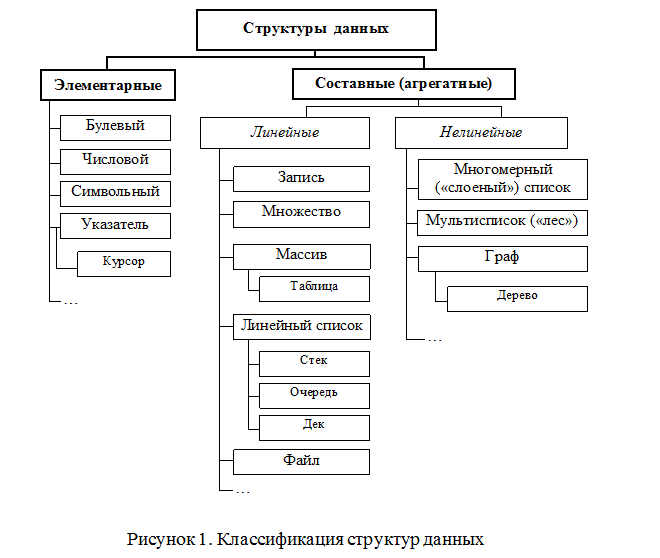
Абстрактный тип данных (АТД) – это математическая модель с совокупностью операторов или операций определенных в рамках данной модели. Операции АТД реализуют функциональное назначения математической модели.

Мы можем разрабатывать алгоритм в терминах АТД, но для реализации алгоритма в конкретном языке программирования необходимо найти способ представления АТД в терминах типов данных и операторов, поддерживаемых данным языком программирования. Для представления АТД используются структуры данных, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом

Составными называются такие структуры данных, составными частями которых являются другие структуры данных - элементарные или в свою очередь составные. Составные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

Важный признак составной структуры данных – характер упорядоченности ее составных частей. По этому признаку структуры можно делить на линейные и нелинейные структуры.

Классификация структур данных по вышеназванным признакам приведена ниже (см. Рисунок 1).



Двусвязная очередь (жарг. дэк, дек от англ. deque — double ended queue; двухсторонняя очередь, очередь с двумя концами) — структура данных, в которой элементы можно добавлять и удалять как в начало, так и в конец. Может быть реализована при помощи двусвязного списка.

## **2.2 Принципы анализа эффективности алгоритма**

Сложность алгоритма – это величина, отражающая порядок величины требуемого ресурса (времени или дополнительной памяти) в зависимости от размерности задачи.

Самый простой способ оценки – экспериментальный, то есть запрограммировать алгоритм и выполнить полученную программу на нескольких задачах, оценивая время выполнения программ. Однако, этот способ имеет ряд недостатков. Во-первых, экспериментальное программирование – это, возможно, дорогостоящий процесс. Во-вторых, необходимо учитывать, что на время выполнения программ влияют следующие факторы:

1) временная сложность алгоритма программы;

2) качество скомпилированного кода исполняемой программы в силу различий в реализации отдельных операторов языка высокого уровня с учетом «оптимизации» компилятора под конкретный процессор;

3) внешние задержки, вызванные работой операционной системы, например, при реализации механизма многозадачности или других программ, работающих в «фоновом» режиме (например, антивирусы);

4) машинные инструкции, используемые для выполнения программы, которые ориентированы на аппаратные особенности архитектуры ЭВМ, например, параллельную обработку линейной последовательности данных.

Наличие трех последних факторов не позволяет применять типовые единицы измерения временной сложности алгоритма (секунды, миллисекунды и т.п.), так как можно получить самые различные оценки для одного и того же алгоритма, если использовать труд разных программистов (которые реализуют алгоритм каждый по-своему), разные компиляторы, операционные системы и разные вычислительные машины.

Таким образом, будем различать время выполнения программы, которое можно измерить в секундах (миллисекундах, аппаратных тактах центрального процессора) и время выполнения соответствующего ей алгоритма, которое будем измерять числом инструкций (элементарных или основных операций), которые необходимо выполнить для получения требуемого результата.

Практически время выполнения алгоритма зависит не только от количества входных данных, но и от их значений, например, время работы некоторых алгоритмов сортировки значительно сокращается, если первоначально данные частично упорядочены, тогда как другие методы оказываются нечувствительными к этому свойству. Чтобы учитывать этот факт, полностью сохраняя при этом возможность анализировать алгоритмы независимо от данных, различают:

1. максимальную сложность Tmax(n), или сложность наиболее неблагоприятного случая, когда алгоритм работает дольше всего;
2. среднюю сложность Tmid(n) – сложность алгоритма в среднем;
3. минимальную сложность Tmin(n) – сложность в наиболее благоприятном случае, когда алгоритм справляется быстрее всего.

**Асимптотические обозначения**

Если  и  положительно определенные функции, то запись  означает, что найдутся константы *c*1 > 0 и *c*2 > 0 и такое *n*0 > 0, что  для всех .

Если , то говорят, что  является асимптотически точной оценкой (asymptotically tight bound) для . На самом деле это отношение симметрично, если: , то .



Рис. 2. Иллюстрация к определению 

Теоретическая оценка временной сложности алгоритма осуществляется с использованием следующих базовых принципов оценки сложности вы-полнения основных алгоритмических конструкций:

1. Время выполнения операторов присваивания, чтения и записи обычно имеет порядок О(1), как всякий линейный оператор, кроме случаев использования внутри них функций и процедур. В этом случае оператор имеет порядок выполнения функции или процедуры.
2. Время выполнения последовательности операторов определяется с помощью правила сумм.
3. Время выполнения условных операторов состоит из времени выполнения условно исполняемых операторов и времени вычисления самого логического выражения.
4. Время выполнения цикла является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла.
5. Для программ, содержащих несколько процедур (функций) (среди которых нет рекурсивных), можно подсчитать общее время выполнения программы путем последовательного нахождения времени выполнения процедур, начиная с той, которая не имеет вызовов других процедур.
6. Если есть рекурсивные процедуры, то нельзя упорядочить все процедуры таким образом, чтобы каждая процедура вызывала только процедуры, время выполнения которых подсчитано на предыдущем шаге. В этом случае мы должны с каждой рекурсивной процедурой связать временную- функцию Т(n), где п определяет объем аргументов процедуры. Затем мы должны получить рекуррентное соотношение для Т(n).

## **2.3 Сортировка простой вставкой**

Сортировка вставками (англ. Insertion sort) — алгоритм сортировки, в котором элементы входной последовательности просматриваются по одному, и каждый новый поступивший элемент размещается в подходящее место среди ранее упорядоченных элементов.

На вход алгоритма подаётся последовательность *n* чисел: *a1, a2, ..., an*. Сортируемые числа также называют ключами. Входная последовательность на практике представляется в виде массива с n элементами. На выходе алгоритм должен вернуть перестановку исходной последовательности *a`1, a`2, ..., a`n*, чтобы выполнялось следующее соотношение *a`1 <= a`2 <= ...<= a`n.*

В начальный момент отсортированная последовательность пуста. На каждом шаге алгоритма выбирается один из элементов входных данных и помещается на нужную позицию в уже отсортированной последовательности до тех пор, пока набор входных данных не будет исчерпан. В любой момент времени в отсортированной последовательности элементы удовлетворяют требованиям к выходным данным алгоритма.

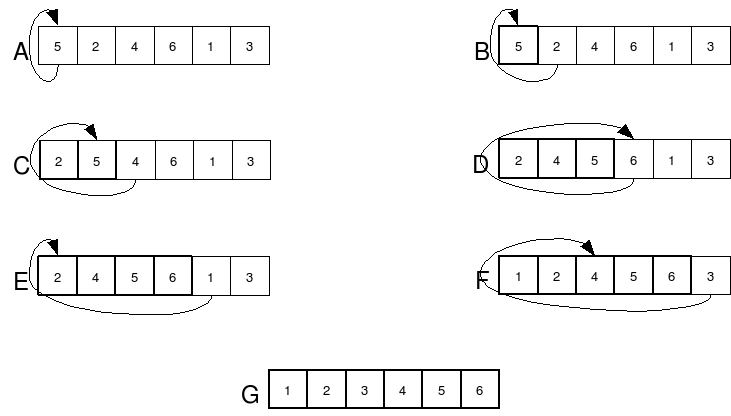


Рис. 3 Пример сортировки вставками

# **Глава 3. Экспериментальная часть**

## **3.1 Листинг программы**

#include <iostream>

#include <conio.h>

#include <cstdlib>

using namespace std;

class Dec{

private:

struct Node{

int c;

Node \*next;

Node \*prev;

};

typedef Node \*PNode;

PNode Head, Tail;

int leght;

PNode CreateNode(int NewInt){

PNode NewNode = new Node;

NewNode->c = NewInt;

NewNode->next = NULL;

NewNode->prev = NULL;

return NewNode;

}

public:

Dec(){

leght = 0;

Head = NULL;

Tail = NULL;

}

bool empty(){

if(Head == NULL){

return 1;

}

else{

return 0;

}

}

void AddL(int Num){

PNode NewNode = CreateNode(Num);

if(Head == NULL){

NewNode->next = NULL;

NewNode->prev = NULL;

Head = NewNode;

Tail = NewNode;

}

else{

NewNode->next = Head;

NewNode->prev = NULL;

Head->prev = NewNode;

Head = NewNode;

}

leght++;

}

void DelL(){

if(Head == NULL){

cout << "List Null" << endl;

return;

}

else{

Head = Head->next;

if(Head != NULL){

Head->prev = NULL;

}

else{

Tail = NULL;

}

leght--;

}

}

void DelR(){

if(Tail == NULL){

cout << "List Null" << endl;

return;

}

else{

Tail = Tail->prev;

if(Tail != NULL){

Tail->next = NULL;

}

else{

Head = NULL;

}

leght--;

}

}

void AddR(int Num){

PNode NewNode = CreateNode(Num);

if(Tail == NULL){

NewNode->next = NULL;

NewNode->prev = NULL;

Head = NewNode;

Tail = NewNode;

}

else{

NewNode->next = NULL;

NewNode->prev = Tail;

Tail->next = NewNode;

Tail = NewNode;

}

leght++;

}

void Enter(){

if(!empty()){

PNode p = Head;

while(p != NULL){

cout << p->c << " ";

p = p->next;

}

cout << endl;

}

}

int ValueL(){

if(!empty()){

return Head->c;

}

}

int ValueR(){

if(!empty()){

return Tail->c;

}

}

void NewDec(){

leght = 0;

Head = NULL;

Tail = NULL;

}

void Sort(){

for(int i = 1;i < leght;i++){

int b = ValueR();DelR();

int j;

for(j = 0;j<=i;j++){

if(b < ValueL() || j==i){

AddL(b);

break;

}

else{

AddR(ValueL());DelL();

}

}

for(int k = j;k > 0;k--){

AddL(ValueR());DelR();

}

}

}

};

int EnterNum;

int main(){

Dec List;

int n;

cout << "n:";cin >> n;

for(int i = 0;i < n;i++){

List.AddL(rand() % 100);

}

List.Enter();

List.Sort();

List.Enter();

}

* 1. **Анализ сложности алгоритма**

В методах добавления AddL() и AddR() производится по 10 операций. В методах удаления DelR() и DelL() производится по 5 операций. В методах возвращающее значение ValueL() и ValueR() производится по 2 операций. Подставив эти значения можно высчитать трудоемкость алгоритма сортировки, который представлен методом Sort().

F(n) = 25n^2 +48n + 2

O(F(n)) = 25n^2

Таблица 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | F(n) | O(F(n)) | N\_op | T(n) |
| 1000 | 25048002 | 25000000 | 12220568 | 0.049 |
| 2000 | 100096002 | 100000000 | 89634757 | 0.174 |
| 3000 | 225144002 | 225000000 | 198344842 | 0.386 |
| 4000 | 400192002 | 400000000 | 345180062 | 0.620 |
| 5000 | 625240002 | 625000000 | 542689902 | 0.997 |
| 6000 | 900288002 | 900000000 | 779768432 | 1.397 |
| 7000 | 1225336002 | 1225000000 | 1053065377 | 1.901 |
| 8000 | 1600384002 | 1600000000 | 1366575227 | 2.547 |
| 9000 | 2020432002 | 2000000000 | 1723004977 | 3.121 |
| 10000 | 2500480002 | 2500000000 | 2130086652 | 3.781 |

Таблица 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | C1=F(n)/T(n) | C2= O(F(n))/ T(n) | C3=F(n)/ N\_op | C4= O(F(n))/ N\_op |
| 1000 | 510204 | 511183 | 2,049659394 | 2,04573142 |
| 2000 | 574713 | 575264 | 2,156149593 | 2,154081631 |
| 3000 | 582901 | 583274 | 2,117168356 | 2,115814216 |
| 4000 | 645161 | 645470 | 2,072579143 | 2,071584772 |
| 5000 | 626880 | 627121 | 2,083731541 | 2,082931689 |
| 6000 | 644237 | 644443 | 2,081278117 | 2,080612316 |
| 7000 | 644397 | 644574 | 2,064532846 | 2,063966726 |
| 8000 | 628190 | 628340 | 2,050196383 | 2,049704452 |
| 9000 | 648830 | 648968 | 2,044388108 | 2,043952063 |
| 10000 | 661200 | 661327 | 2,047210754 | 2,046817763 |

По таблице 2 построим графики.

Рис. 4 График 1

Рис. 5 График 2

* 1. **Особые случаи**

Программа совершит больше всего операций при обратно упорядоченной последовательности.

Таблица 3

|  |  |  |
| --- | --- | --- |
| n | N\_op(1) | N\_op(2) |
| 1000 | 12220568 | 23970008 |
| 2000 | 89634757 | 95940008 |
| 3000 | 198344842 | 215910008 |
| 4000 | 345180062 | 383880008 |
| 5000 | 542689902 | 599850008 |
| 6000 | 779768432 | 863820008 |
| 7000 | 1053065377 | 1175790008 |
| 8000 | 1366575227 | 1535760008 |
| 9000 | 1723004977 | 1943730008 |
| 10000 | 2130086652 | 2399700008 |

Рис. 6 График 3

Программа совершит наименьшее количество операций при полностью упорядоченной последовательностью. Таблица 4

|  |  |  |
| --- | --- | --- |
| n | N\_op(1) | N\_op(3) |
| 1000 | 12220568 | 89912 |
| 2000 | 89634757 | 179912 |
| 3000 | 198344842 | 269912 |
| 4000 | 345180062 | 359912 |
| 5000 | 542689902 | 449912 |
| 6000 | 779768432 | 539912 |
| 7000 | 1053065377 | 629912 |
| 8000 | 1366575227 | 719912 |
| 9000 | 1723004977 | 809912 |
| 10000 | 2130086652 | 899912 |

Рис. 7 График 4

# **Глава 4. Вывод**

В ходе данной работы была подсчитана трудоемкость алгоритма сортировки простая вставка. В качестве структуры данных использовался линейный список – дек. На основе навыков анализа эффективности алгоритмов, теоретически была получена верхняя асимптотическая оценка алгоритма:

O(F(n)) = n^2

Проанализировав графики 1 и 2 можно сказать, что C1=F(n)/T(n) примерно равно C2= O(F(n))/ T(n), а C3=F(n)/ N\_op примерно равно C4= O(F(n))/ N\_op. Также из графика 1 видно, что время работы алгоритма растет не равномерно. Это доказывает, что на время работы алгоритма влияют посторонние программы.

Исходя из графиков 3 и 4 можно сделать вывод, что время выполнения алгоритма зависит от входных данных: чем большее множество нужно отсортировать, тем большее время потребуется для выполнения сортировки. Также на время выполнения влияет исходная упорядоченность заданной последовательности элементов. Также с их помощью можно получить нижнюю асимптотическую оценку алгоритма сортировки:



# **Список литературы**

* Ахо А. В., Хопкрофт Д. Э., Ульман Д. Д. Структуры данных и алгоритмы = Data structures and algorithms / Под ред. А. А. Минько. — М.: Вильямс, 2000.
* Кнут Д. Э. 5.2.1 Сортировка путём вставок // Искусство программирования. Том 3. Сортировка и поиск = The Art of Computer Programming. Volume 3. Sorting and Searching / под ред. В. Т. Тертышного (гл. 5) и И. В. Красикова (гл. 6). — 2-е изд. — Москва: Вильямс, 2007
* Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. 2.1. Сортировка вставкой // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 3-е изд. — М.: Вильямс, 2013.
* Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004.
* Филатов В.В. : Структуры и алгоритмы обработки данных(Анализ эффективности алгоритмов) – М.:МГУПИ, 2009

**Использованные источники**

* Сортировка вставками. [Электронный ресурс] Режим доступа: <http://algolist.manual.ru/sort/insert_sort.php>
* Сортировка вставками. [Электронный ресурс] Режим доступа: https://habr.com/ru/post/415935/
* Реализация сортировки вставками на Pascal и C++ [Электронный ресурс] Режим доступа: <http://kvodo.ru/sortirovka-vstavkami-2.html>