ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение и администрирование информационных систем»

Учебно-методическое пособие

по дисциплине

«АЛГОРИТМЫ, ПОСТРОЕНИЕ И АНАЛИЗ»

Автор

Криворучко А.В.

Ростов-на-Дону, 2018

Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения направления 09.03.04 «Программная инженерия».

Авторы

доцент, к.ф.-м.н.,
каф. ПОВТиАС
Криворучко А.В.

Оглавление

1. ЛАБОРАТОРНАЯ РАБОТА №1: ПОЛУСТАТИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

1.1. Цель работы

- исследовать и изучить полустатические структуры данных (на примере стеков, реализованных с помощью массивов);

- овладеть навыками разработки алгоритмов и написания программ на по исследованию стеков на языке программирования C++;

1.2. Теория

Порядок выполнения работы

- ознакомиться с краткой теорией и примерами решения задач, относящихся к работе со стеками;

- ответить на контрольные вопросы и по знанию теории;

- получить задание на выполнение конкретного варианта лабораторной работы у преподавателя и выполнить его;

- написать и отладить программу решения задачи на языке C++;

- подготовить отчет по лабораторной работе и защитить его у преподавателя.

Содержание отчета по ЛР

- наименование ЛР и ее цель;

- задание на ЛР согласно варианту;

- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;

- результаты работы программы.

Краткая теория

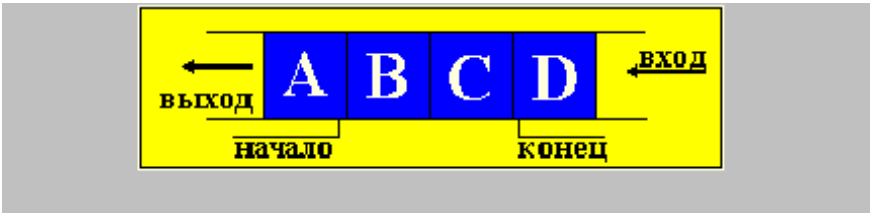
Понятие очереди всем хорошо известно из повседневной жизни. Элементами очереди в общем случае являются заказы на то или иное обслуживание: выбить чек на нужную сумму в кассе магазина; получить нужную информацию в справочном бюро, выполнить очередную операцию по обработке детали на данном станке в автоматической линии и т.д.

В программировании имеется структура данных, которая называется очередь. Эта структура данных используется, например, для моделирования реальных очередей с целью определения их характеристик (средняя длина очереди, время пребывания заказа в очереди и т.п.) при данном законе поступления заказов и дисциплине их обслуживания.

По своему существу очередь является полустатической структурой - с течением времени и длина очереди, и набор образующих ее элементов могут изменяться.

Различают два основных вида очередей, отличающихся по дисциплине обслуживания находящихся в них элементов:

1. При первой из дисциплин заказ, поступивший в очередь первым, выбирается первым для обслуживания (и удаляется из очереди). Эту дисциплину обслуживания принято называть **FIFO** (*First input-First output*, т.е. первый пришел - первый ушел). Очередь открыта с обеих сторон.



2. Вторую дисциплину принято называть **LIFO** (*Last input - First output*, т.е. последний пришел - первый ушел), при которой на обслуживание первым выбирается тот элемент очереди, кото-

рый поступил в нее последним. Очередь такого вида в программировании принято называть СТЕКОМ (магазином) - это одна из наиболее употребительных структур данных, которая оказывается весьма удобной при решении различных задач.

В силу указанной дисциплины обслуживания, в стеке доступна единственная его позиция, которая называется ВЕРШИНОЙ стека - эта позиция, в которой находится последний по времени поступления в стек элемент. Когда мы заносим новый элемент в стек, то он помещается поверх вершины и теперь уже сам находится в вершине стека. Выбрать элемент можно только из вершины стека; при этом выбранный элемент исключается из стека, а в его вершине оказывается элемент, который был занесен в стек перед выбранным из него элементом (структура с ограниченным доступом к данным).

ОПЕРАЦИИ НАД СТЕКАМИ:

- PUSH (S, x) - занесение элемента в стек, где s - название стека, x - элемент, который заносится в стек;
- POP (S) - выборка элемента из стека. При выборке элемент помещается в рабочую область памяти, где он используется;
- EMPTY (S) - проверка стека на пустоту (true - пуст, false - не пуст);
- STACKTOP (S) - чтение верхнего элемента без его удаления.
- FULL (S) – проверка стека на переполнение (в случае, если стек реализован с помощью массива.

1.3. Задания

Ввести символы, формируя из них стек.

Варианты

1. Поменять местам первый и последний элементы стека.

2. Развернуть стек, т.е. сделать "дно" стека вершиной, а вершину - "дном"

3. Удалить элемент, находящийся в середине стека , если число элементов нечетное, или 2 средних элемента, если число элементов четное.

4. Удалить каждый второй элемент стека

5. Вставить символ '*' в середину стека, если число элементов четное, или после среднего элемента, если число элементов нечетное.

6. Найти минимальный элемент и вставить после него 0.

7. Найти максимальный элемент и вставить после него 0

8. Удалить минимальный элемент.

9. Удалить все элементы, равные первому.

10. Удалить все элементы, равные последнему.

11. Удалить максимальный элемент.

12. Найти минимальный элемент и вставит на его место 0.

Вывести полученный стек на экран.

Составить отчет по лабораторной работе и защитить его у преподавателя.

2. ЛАБОРАТОРНАЯ РАБОТА №2: СПИСКОВЫЕ СТРУКТУРЫ ДАННЫХ

2.1. Цель работы

- исследовать и изучить списковые структуры данных и их основные процедуры;

- овладеть умениями и навыками написания программ по исследованию списковых структур данных и их основных процедур на языке программирования C++;

2.2. Теория

Порядок выполнения работы

- ознакомиться с краткой теорией и примерами решения задач, относящихся к работе со списковыми структурами данных;

- ответить на контрольные вопросы и получить оценку по знанию теории;

- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;

- написать и отладить программу решения задачи на языке C++;

- подготовить отчет по лабораторной работе и защитить его у преподавателя.

Содержание отчета по ЛР

- наименование ЛР и ее цель;

- задание на ЛР согласно варианту;

- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;

- результаты работы программы.

Краткая теория

В лабораторном занятии №1 рассматривались только статические программные объекты. Этим термином обозначаются объекты, которые порождаются непосредственно перед выполнением программы, существуют в течение всего времени ее выполнения и размер значений которых не изменяется по ходу выполнения программы.

Поскольку статические объекты порождаются до выполнения программы и размер их значений можно выделить еще на этапе трансляции исходного текста программы на языке машины.

Однако использование при программировании только статических объектов может вызвать определенные трудности, особенно с точки зрения получения эффективной машинной программы, а такая эффективность бывает чрезвычайно важной

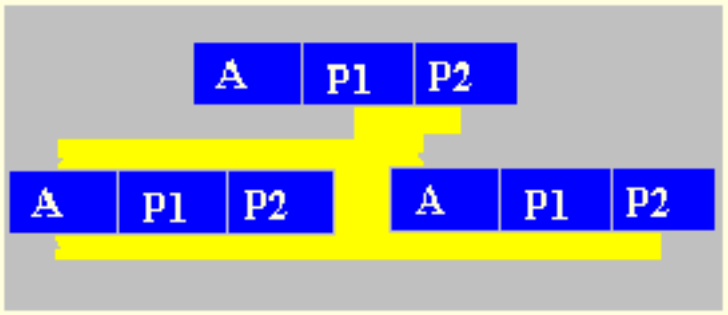
при решении ряда задач. Дело в том, что иногда мы заранее не знаем не только размера значения того или иного программного объекта, но даже и того, будет ли существовать этот объект или нет. Такого рода программные объекты, которые возникают уже в процессе выполнения программы, называют динамическими объектами.

Динамические структуры данных имеют две особенности :

1. Заранее не определимо количество элементов в структуре;

2. Элементы динамической структуры не имеют жесткой линейной упорядоченности. Они могут быть разбросаны по памяти.

Чтобы связать элементы динамической структуры между собой, в состав элемента помимо информационного поля входят поля указателей (связок с другими элементами структуры).



p1, p2 - указатели, содержащие адреса элементов, с которыми данный элемент связан.

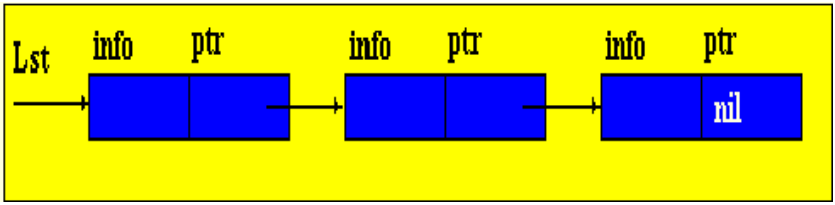
Наиболее распространенными динамическими структурами являются связанные списки. С точки зрения логического представления различают *линейные* и *нелинейные* списки. В линейных списках связи строго упорядочены. Указатель предыдущего элемента дает адрес последующего элемента или наоборот.

К линейным спискам относятся *односвязные* и *двусвязные* списки.

К нелинейным - многосвязные списки.

Элемент списка в общем случае представляет собой комбинацию поля записи (информационного поля) и одного или нескольких указателей.

Линейные однонаправленные списки (односвязные списки)



Под односвязными списками понимают упорядоченную последовательность элементов, каждый из которых имеет 2 поля:

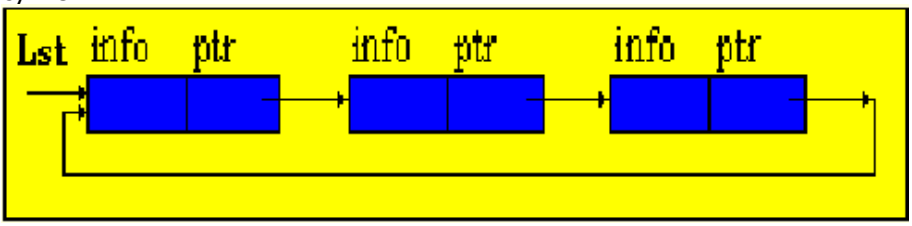
- информационное поле *info* и
- поле указателя *ptr* .

Особенностью указателя является то, что он дает только адрес последующего элемента списка. У однонаправленных списков поле указателя последнего элемента в списке является пустым nil.

Lst - указатель начала списка. Он представляет список как единое целое. Иногда список может быть пустым, т.е. в данном списке нет ни одного элемента. В этом случае *lst* = nil.

Кольцевые односвязные списки

Пример кольцевого списка представлен на следующем рисунке.



На этом рисунке, список замыкается в своеобразное "кольцо": двигаясь по ссылкам, можно от последнего элемента списка переходить к заглавному элементу. В связи с этим списки подобного рода называют кольцевыми списками.

ОПЕРАЦИИ НАД ОДНОСВЯЗНЫМИ СПИСКАМИ:

1. Вставка элемента в начало односвязного списка.
2. Удаление элемента из начала односвязного списка
3. Вставка элемента в список
4. Удаление элемента из односвязного списка

ОПЕРАЦИИ НАД КОЛЬЦЕВЫМИ СПИСКАМИ:

1. Вставка элемента в кольцевой список
2. Удаление элемента из кольцевого списка

2.3. Задания

Односвязные списки

1. Написать программу передвижения элемента на n позиций.
2. Создать копию списка.
3. Добавить элемент в начало списка.
4. Склеить два списка.
5. Удалить *n*-ый элемент из списка.
6. Вставить элемент после n-го элемента списка.
7. Создать список содержащий элементы общие для двух списков.
8. Упорядочить элементы в списке по возрастанию.
9. Удалить каждый второй элемент списка.
10. Удалить каждый третий элемент списка.
11. Упорядочить элементы списка по убыванию.
12. Очистить список.

Кольцевые списки

13. Дан кольцевой список, содержащий 20 фамилий игроков футбольной команды. Разбить игроков на 2 группы по 10 человек. Во вторую группу попадает каждый 2-й человек.
14. Даны 2 кольцевых списка, содержащие фамилии спортсменов двух фехтовальных команд. Произвести жеребьевку. В первой команде выбирается каждый *n*-й игрок, а во второй - каждый *m*-й.
15. Задача Джозефуса: n воинов из одного войска убивают каждого m-го из другого. Требуется определить номер k начальной позиции воина, который должен будет остаться последним.
16. Даны 2 кольцевых списка, содержащие фамилии участников лотереи и наименования призов. Выиграет N человек (каждый К-й). Число для пересчета призов - t. Вывести фамилии выигравших.
17. Даны 2 списка, содержащих фамилии учащихся и номера экзаменационных билетов. Число пересчета для билетов - E, для учащихся - K. Определить номера билетов, вытащенных учащимися.
18. Дан список, содержащий перечень товаров. Из элементов 1-го списка (товары изготовленные фирмой SONY) создать новый список.

19. Даны 2 списка, содержащие фамилии студентов 2-х групп. Перевести L студентов из 1-й группы во вторую. Число пересчета -K.

20. Даны 2 списка, содержащие перечень товаров, производимых концернами BOSH и FILIPS. Создать список товаров, выпускаемых как одной, так и другой фирмой.

21. Даны 2 списка, содержащие фамилии футболистов основного состава команды и запасного. Произвести K замен.

22. Даны 2 списка, содержащие фамилии солдат 1-го и 2-го взводов. Во время атаки M человек из 1-го взвода погибли. Произвести пополнение солдатами 2-го взвода.

23. Даны 2 списка, содержащие перечень товаров и фамилии покупателей. Каждый N-й покупатель покупает M-й товар. Вывести список покупок.

24. Даны 2 списка, содержащие наименования товаров, выпускаемых фирмами SONY и SHARP. Создать список конкурирующих между собой товаров.

3. ЛАБОРАТОРНАЯ РАБОТА №3: БИНАРНЫЕ ДЕРЕВЬЯ (СОЗДАНИЕ И ОБХОД

3.1. Цель работы

- исследовать и изучить основные процедуры, используемые при работе с бинарными (двоичными) деревьями;

- овладеть умениями и навыками написания программ по исследованию бинарных деревьев на языке программирования C++.

3.2. Теория

Порядок выполнения работы

- ознакомиться с краткой теорией и примерами решения задач, относящихся к работе с бинарными деревьями;

- ответить на контрольные вопросы по знанию теории;

- получить задание на выполнение конкретного варианта лабораторной работы у преподавателя и разработать алгоритм решения задачи;

- написать и отладить программу решения задачи на языке C++;

- подготовить отчет по лабораторной работе и защитить его у преподавателя.

Содержание отчета по ЛР

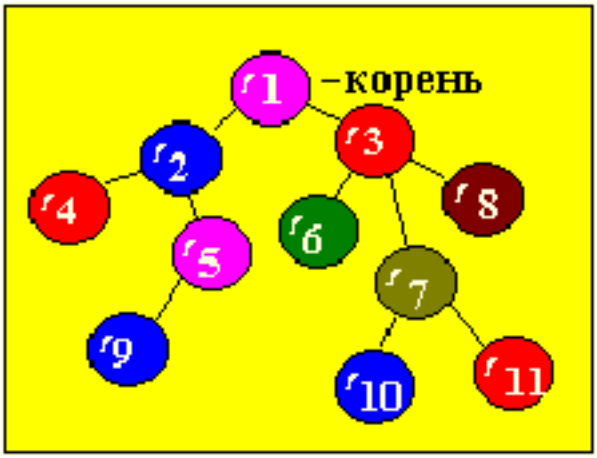
- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

Краткая теория

Дерево - это нелинейная связанная структура данных, характеризуемая следующими признаками:

- · дерево имеет один элемент, на который нет ссылок от других элементов. Этот элемент, или "узел", называется корнем дерева;
- · в дереве можно обратиться к любому элементу путем прохождения конечного числа ссылок (указателей);
- · каждый элемент дерева связан только с одним предыдущим элементом.

Каждый узел дерева может быть промежуточным (элементы 2,3,5,7) либо терминальным ("листом" дерева) (элементы 4,9,10,11,8,6). Характерной особенностью терминальных узлов является отсутствие ветвей.



Элемент *Y*, находящийся непосредственно ниже элемента *X*, называется непосредственным потомком *X*, если *X* находится на

уровне *i* , то говорят, что *Y* лежит на уровне *i+1*. Считается, что корень лежит на уровне 0.

Число непосредственных потомков элемента называется его степенью исхода, в зависимости от степени исхода узлов деревья классифицируют:

A. Если степень исхода узлов - *M*, то дерево называется *M*-арным ;

B. Если степень исхода узлов - *M* или *0*, то - полное *M*-арное дерево;

C. Если степень исхода узлов дерева равна *2*, то дерево называется бинарным ;

D. Если степень исхода равна *2* или *0*, то - полное бинарное дерево.

Особенно важную роль играют бинарные деревья, поэтому далее мы будем рассматривать их более подробно.

Представление деревьев в памяти ЭВМ

Деревья наиболее удобно представлять в памяти ЭВМ в виде связанных нелинейных списков. Элемент должен содержать *INFO*-поле, где содержится характеристика узла. Следующее поле определяет степень исхода узла и количество полей указателей равное степени исхода. Каждый указатель элемента ориентирует данный элемент-узел с его сыновьями. Узлы, в которые входят стрелки от исходного элемента, называются его сыновьями.

INFO-поле содержит два поля : поле записи (*rec*) и поле ключа (*key*). Ключ задается числом, по ключу определяют место элемента в дереве.

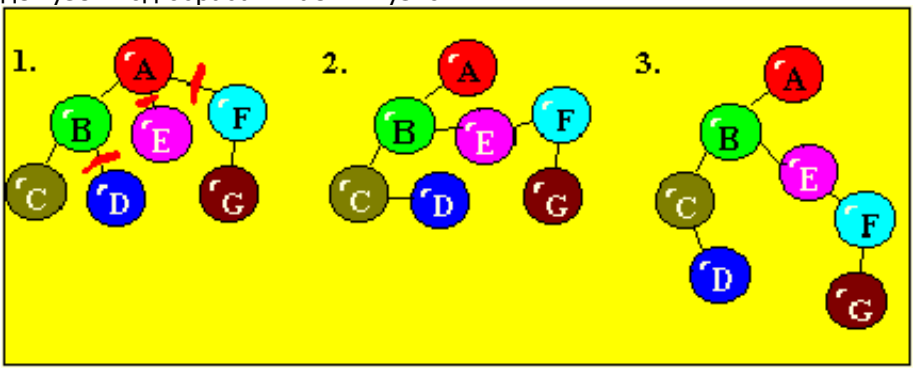
Представление деревьев в Эвм

Сведение *m*-арного дерева к бинарному

1. В каждом узле дерева отсекают все ветви, кроме крайних левых, соответствующих старшим сыновьям;

2. Соединяют горизонтальными линиями сыновей одного родителя (узла);

3. Старшим сыном в каждом узле полученной структуры будет узел под обрабатываемым узлом.

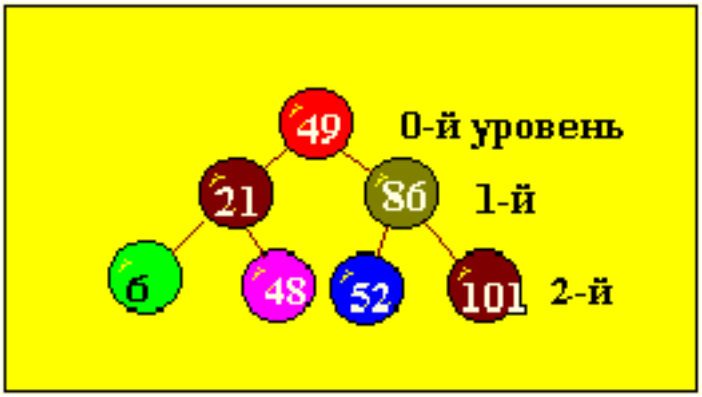


Построение бинарных деревьев. Согласно представлению деревьев в памяти ЭВМ каждый элемент (узел бинарного дерева) будет записью, содержащей четыре поля. Значением этих полей будут соответственно

- ключ записи,
- ссылка
- · на элемент влево-вниз,
- · на элемент вправо-вниз и
- · на текст записи.

При построении необходимо помнить, что левый сын имеет ключ меньше чем отец (родитель). Значение ключа правого сына больше значения ключа отца.

Если узлы дерева имеют значения 6, 21, 48, 49, 52, 86, 101, то бинарное дерево может иметь вид, изображенный на рисунке ниже. Это упорядоченное бинарное дерево с минимальным числом уровней. Идеально сбалансированное дерево - это дерево, в котором левое и правое поддеревья имеют уровни, отличающиеся не больше, чем на единицу.



3.3. Контрольные вопросы

1. Описать процедуру или функцию, которая:

a) присваивает параметру *E* запись из самого левого листа непустого дерева *T* (лист-вершина, из которого не выходит ни одной ветви);

b) определяет число вхождений записи *E* в дерево *T*.

2. Вершины дерева вещественные числа. Описать процедуру или функцию, которая:

a) вычисляет среднее арифметическое всех вершин дерева;

b) добавляет в дерево вершину со значением, вычисленным в предыдущей процедуре (функции).

3. Записи вершин дерева - вещественные числа. Описать процедуру, которая выбирает все вершины с отрицательными записями и строит из них новое дерево.

4. Записи вершин дерева - вещественные числа. Описать процедуру или функцию, которая:

a) находит максимальное или минимальное значение записей вершин непустого дерева;

b) печатает записи из всех листьев дерева.

5. Описать процедуру или функцию, которая:

a) определяет, входит ли вершина с записью *E* в дерево *T*;

b) если такая запись не найдена, то она добавляется.

6. Описать процедуру или функцию, которая:

a) находит в непустом дереве *T* длину (число ветвей) пути от корня до ближайшей вершины с записью *E*; если *E* не входит в *T*, то за ответ принять - *1*.

b) определяет максимальную глубину непустого дерева T, т.е. число ветвей в самом длинном из путей от корня дерева до листьев.

7. Описать процедуру COPY(*T,T1*), которая строит *T1* - копию дерева *T*.

8. Описать процедуру EQUAL(*T1,T2*), проверяющую на равенство деревья *T1* и *T2* (деревья равны, если ключи и записи вершин одного дерева равны соответственно ключам и записям другого дерева).

9. Описать процедуру, которая:

a) присваивает переменной *b* типа *char* значение:

K - если вершина - корень,

П - если вершина - промежуточная вершина,

Л - если вершина - лист;

b) распечатывает атрибуты всех вершин дерева.

10. Описать процедуру, которая:

a) находит максимальное или минимальное значение записей листьев непустого дерева;

b) добавляет элемент с данной записью в дерево.

11. Описать процедуру или функцию, которая:

a) вставляет узел с записью E в дерево, если ранее такой не было;

b) считает и выдает на экран сумму значений всех ключей, если такая запись есть.

12. Описать процедуру или функцию, которая:

a) печатает запись, встречающуюся в дереве один раз;

b) печатает запись, встречающееся в дереве максимальное число раз.

4. ЛАБОРАТОРНАЯ РАБОТА №4: ИССЛЕДОВАНИЕ МЕТОДОВ ЛИНЕЙНОГО И БИНАРНОГО ПОИСКА

4.1. Цель работы

- изучить методы линейного, бинарного и идексно-последовательного поиска.

- овладеть навыками написания программ для методов линейного, бинарного и индексно-последовательного поиска на языке программирования C++.

4.2. Теория

Порядок выполнения работы

- ознакомиться с краткой теорией и примерами решения задач, относящихся к исследованию методов линейного, бинарного и индексно-последовательного поиска;

- ответить на контрольные вопросы и получить оценку по знанию теории;

- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;

- написать и отладить программу решения задачи на языке C++;

- составить отчет по лабораторной работе и защитить его у преподавателя.

Содержание отчета по ЛР

- наименование ЛР и ее цель;

- задание на ЛР согласно варианту;

- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;

- результаты работы программы.

Краткая теория

Поиск – это действие наиболее часто встречающееся в программировании. Он же представляет собой идеальную задачу, на которой можно испытывать различные структуры данных по мере их появления. Существует несколько основных "вариаций этой темы", и для них создано много различных алгоритмов. При дальнейшем рассмотрении мы исходим из такого принципиального допущения: группа данных, в которой необходимо отыскать

заданный элемент, фиксирована. Будем считать, что множество из *N* элементов задано, скажем, в виде такого массива

a: ARRAY[*0..N-1*] OF item

Обычно тип item описывает запись с некоторым полем, выполняющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному "аргументу поиска" *x*. Полученный в результате индекс *i*, удовлетворяющий условию *a[i].key=x*, обеспечивает доступ к другим полям обнаруженного элемента. Так как нас интересует в первую очередь сам процесс поиска, а не обнаруженные данные, то мы будем считать, что тип *item* включает только ключ, т.е. он есть ключ (*key*).

Линейный поиск

Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход - простой последовательный просмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Такой метод называется линейным поиском. Условия окончания поиска таковы:

1. Элемент найден, т.е. *a[i] = x*.
2. Весь массив просмотрен и совпадения не обнаружено.

Это дает нам линейный алгоритм:

```
i := 0;
WHILE (i < N) AND (a[i] <> x) DO
  i := i+1 ;
END;
```

Обратите внимание, что порядок элементов в логическом выражении имеет существенное значение. Инвариант цикла, т.е. условие, выполняющееся перед каждым увеличением индекса i, выглядит так:

$(0 \le i < N)$ AND $(A_k : 0 \le k < i : a_k \neq x)$

Он говорит, что для всех значений *k*, меньших чем *i*, совпадения не было. Отсюда и из того факта, что поиск заканчивается только в случае ложности условия в заголовке цикла, можно вывести окончательное условие его окончания:

$((i = N) OR (a_i = x)) AND (A_k : 0 \le k < i : a_k \neq x)$

Это условие не только указывает на желаемый результат, но из него же следует, что если элемент найден, то он найден вместе с минимально возможным индексом, т.е. это первый из таких элементов. Равенство *i = N* свидетельствует, что совпадения не существует.

Совершенно очевидно, что окончание цикла гарантировано, поскольку на каждом шаге значение *i* увеличивается, и, следовательно, оно, конечно же, достигнет за конечное число шагов предела *N*; фактически же, если совпадения не было, это произойдет после *N* шагов.

Ясно, что на каждом шаге требуется увеличивать индекс и вычислять логическое выражение. А можно ли эту работу упростить и таким образом убыстрить поиск ?

Единственная возможность - попытаться упростить само логическое выражение, ведь оно состоит из двух членов. Следовательно, единственный шанс на пути к более простому решению - сформулировать простое условие, эквивалентное нашему сложному. Это можно сделать, если мы гарантируем, что совпадение всегда произойдет. Для этого достаточно в конец массива поместить дополнительный элемент со значением *x*. Назовем такой вспомогательный элемент "барьером", ведь он охраняет нас от перехода за пределы массива. Теперь массив будет описан так:

a: ARRAY[0..N] OF INTEGER

и алгоритм линейного поиска с барьером выглядит следующим образом:

```
a[N] := x;
i := 0;
WHILE a[i] <> x DO
  i := i+1;
END;
```

Результирующее условие, выведенное из того же инварианта, что и прежде:

$(a_i=x)$ AND $(A_k : 0 \pounds k < i : a_k \neq x)$

Ясно, что равенство $i = N$ свидетельствует о том, что совпадения (если не считать совпадения с барьером) не было.

Поиск делением пополам (двоичный поиск).

Совершенно очевидно, что других способов убыстрения поиска не существует, если, конечно, нет еще какой-либо информации о данных, среди которых идет поиск. Хорошо известно, что поиск можно сделать значительно более эффективным, если данные будут упорядочены. Вообразите себе телефонный справочник, в котором фамилии не будут расположены по порядку. Это нечто совершенно бесполезное! Поэтому мы приводим алгоритм, основанный на знании того, что массив a упорядочен, т.е. удо-

влетворяет условию

$A_k : 1£ k < N : a_{k-1} ¹ a_k$

Основная идея - выбрать случайно некоторый элемент, предположим a[m], и сравнить его с аргументом поиска x. Если он равен x, то поиск заканчивается, если он меньше x, то мы заключаем, что все элементы с индексами, меньшими или равными m, можно исключить из дальнейшего поиска; если же он больше x, то исключаются индексы больше и равные m. Это соображение приводит нас к следующему алгоритму (он называется "поиском делением пополам"). Здесь две индексные переменные L и R отмечают соответственно левый и правый конец секции массива a, где еще может быть обнаружен требуемый элемент.

```
L := 0;
R := N-1;
found := FALSE;
WHILE (L < R) AND NOT found DO
   m := любое значение между L и R;
   IF a[m] = x THEN found := TRUE;
   IF a[m] < x THEN L := m+1
   ELSE R := m-1;
   ENDIF;
ENDWHILE;
```

Инвариант цикла, т.е. условие, выполняющееся перед каждым шагом, таков:

$(L £ R) AND (A_k : 0 £ k < L : a_k < x) AND (A_k : R < k < N : a_k > x)$

из чего выводится результат

$found OR ((L > R) AND (A_k : 0 £ k < L : a_k < x) AND (A_k : R < k < N : a_k > x))$

откуда следует

$(a_m = x) OR (A_k : 0 £ k < N : ak ¹ x)$

Выбор *m* совершенно произволен в том смысле, что корректность алгоритма от него не зависит. Однако на его эффективность выбор влияет. Ясно, что наша задача - исключить на каждом шагу из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В результате максимальное число сравнений равно *log N*, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое

число сравнений - *N/2*.

Эффективность можно несколько улучшить, поменяв местами заголовки условных операторов. Проверку на равенство можно выполнять во вторую очередь, так как она встречается лишь единожды и приводит к окончанию работы. Но более существенно следующее соображение: нельзя ли, как и при линейном поиске, отыскать такое решение, которое опять бы упростило условие окончания. И мы действительно находим такой быстрый алгоритм, как только отказываемся от наивного желания закончить поиск при фиксации совпадения. На первый взгляд это кажется странным, однако при внимательном рассмотрении обнаруживается, что выигрыш в эффективности на каждом шаге превосходит потери от сравнения с несколькими дополнительными элементами. Напомним, что число шагов в худшем случае - *log N*. Быстрый алгоритм основан на следующем инварианте:

$(A_k : 0 \le k < L : a_k < x)$ AND $(A_k : R \le k < N : a_k \ge x)$

причем поиск продолжается до тех пор, пока обе секции не "накроют" массив целиком.

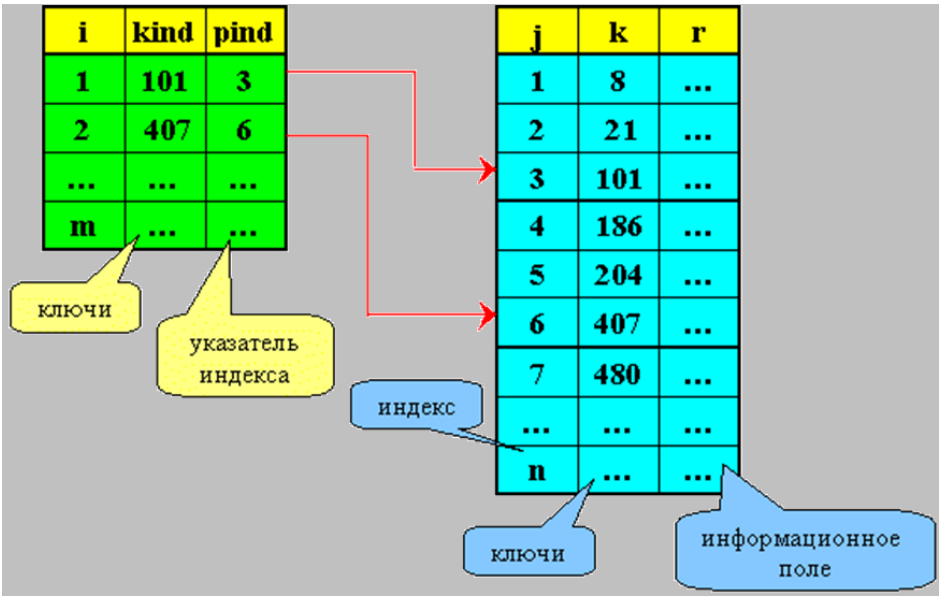
```
L := 0;
R := N;
  WHILE L < R DO
    m := (L+R) DIV 2;
    IF a[m] < x THEN L := m+1
    ELSE R := m ;
  END
END
```

Условие окончания - *L < R*, но достижимо ли оно? Для доказательства этого нам необходимо показать, что при всех обстоятельствах разность *R-L* на каждом шаге убывает. В начале каждого шага *L < R*. Для среднего арифметического *m* справедливо условие *L < m < R*. Следовательно, разность действительно убывает, ведь либо *L* увеличивается при присваивании ему значения *m+1*, либо *R* уменьшается при присваивании значения m. При L = R повторение цикла заканчивается. Однако наш инвариант и условие *L = R* еще не свидетельствуют о совпадении. Конечно, при *R = N* никаких совпадений нет. В других же случаях мы должны учитывать, что элемент *a[R]* в сравнениях никогда не участвует. Следовательно, необходима дополнительная проверка на равенство *a[R] = x*. В отличие от первого нашего решения приведенный алгоритм, как и в случае линейного поиска, находит сов-

падающий элемент с наименьшим индексом.

Еще одним вариантом убыстрения поиска в случае упорядоченности данных является индексно-последовательный поиск. При таком поиске организуется две таблицы: таблица данных со своими ключами - упорядоченных по возрастанию, и таблица ***индексов***, которая тоже состоит из ключей данных, но эти ключи взяты из основной таблицы через определенный интервал. Другими словами, при прохождении упорядоченной таблицы мы сравниваем с ключом элементы не последовательно, а через определенный интервал, то есть задаем некоторый шаг поиска. Когда на очередном шаге поиска предыдущий элемент меньше значения ключа, а следующий элемент больше значения ключа, то устанавливаются соответственно нижняя low (kind<key) и верхняя hi (kind>key) границы в основной таблице. Между этими границами по основной таблице будет соответственно произвен уже обычный последовательный поиск.

Таблицы индексно-последовательного поиска



В псевдокоде алгоритм индексно-последовательного поиска следующий:

```
i = 1
while (i <= m) and (kind(i) <= key) do
        i=i+1
endwhile
if i = 1 then low = 1
           else low = pind(i-1)
endif
if i = m+1 then hi = n
                    else hi = pind(i)-1
endif
for j = low to hi
        if key = k(j) then
                          search = j
                           return
        endif
next j
search = 0
return
```

В принципе, для достижения наибольшей эффективности поиска при решении конкретных задач пользователь может задавать какой угодно шаг.

4.3. Задания

1. Найти наименьший элемент в упорядоченном массиве *A* с помощью линейного, бинарного и индексно-последовательного поиска.

2. Найти элементы в упорядоченном массиве *A*, которые больше *30*, с помощью линейного, бинарного и индексно-последовательного поиска.

3. Вывести на экран все числа массива A кратные *3 (3,6,9,...)* с помощью линейного, бинарного и индексно-последовательного поиска.

4. Найти все элементы, модуль которых больше *20* и меньше 50 в упорядоченной табли- це, с помощью с помощью ли-

нейного, бинарного и индексно-последовательного поиска.

5. Вывести на экран все числа упорядоченного массива *A* кратные *4 (4,8,...)* с помощью помощью линейного, бинарного и индексно-последовательного поиска.

6. Вывести на экран сообщение, каких чисел больше относительно *50* в упорядоченной таблице с помощью линейного, бинарного и индексно-последовательного поиска.

7. Найти элемент в упорядоченном массиве *A* и найти число сравнений помощью линейного, бинарного и индексно-последовательного поиска.

8. Поиск элементов случайным образом помощью линейного, бинарного и индексно-последовательного поиска.

9. Дан список номеров машин *(345, 368, 876, 945, 564, 387, 230)*, найти, на каком месте стоит машина с заданным номером, линейный, бинарный и индексно-последовательный поиск.

10. Поиск каждого кратного двум элемента в упорядоченном массиве помощью линейного, бинарного и индексно-последовательного поиска.

11. Найти элемент с заданным ключом и число сравнений для его нахождения в упорядоченном массиве *A* с помощью линейного, бинарного и индексно-последовательного поиска.

12. Найти элемент с ключом, равным сумме индексов упорядоченного массива *A* с помощью линейного, бинарного и индексно-последовательного поиска.

5. ЛАБОРАТОРНАЯ РАБОТА №5: ИССЛЕДОВАНИЕ МЕТОДОВ ОПТИМИЗАЦИИ ПОИСКА

5.1. Цель работы

- исследовать и изучить методы поиска с перемещением в начало и транспозицией;
- овладеть умениями и навыками написания на C++ про-

грамм по исследованию методов поиска с перемещением в начало и транспозицией.

5.2. Теория

Порядок выполнения работы

- ознакомиться с краткой теорией и примерами решения задач, относящихся к методам оптимизации поиска;
- ответить на контрольные вопросы и получить оценку по знанию теории;
- получить задание на выполнение конкретного варианта лабораторной работы и выполнить его;
- написать и отладить программу решения задачи на языке C++;
- составить отчет по лабораторной работе и защитить его у преподавателя.

Содержание отчета по ЛР

- наименование ЛР и ее цель;
- задание на ЛР согласно варианту;
- листинг приложения, обеспечивающей успешное решение студентом полученного варианта задачи;
- результаты работы программы.

Краткая теория

Поиск (*search*) является одной из основ обработки данных в ЭВМ. Его назначение состоит в том, чтобы по заданному аргументу найти среди массива данных те данные, которые соответствуют этому аргументу или определить, что этих данных нет. Если этих данных нет, добавить их в массив данных.

Набор любых данных будем называть таблицей или файлом. Каждое данное или элемент структуры отличается каким-то признаком от других данных. Этот признак называется ключом. Ключ может быть уникальным, т.е. в таблице только одно данное с таким ключом, иначе уникальные ключи называются первичным ключом.

Вторичный ключ в одной таблице может повторяться, но по нему тоже можно организовать поиск. Ключи данных собраны в одном месте (таблице).

Ключи, которые выделе- ны из таблицы данных и орга-

низованы в свой файл, называются внешним ключами. Если ключ в записи, то он называется внутренним ключом.

Поиском по заданному аргументу называется **алгоритм**, определяющий соответствие ключа данного с заданным аргументом. Результатом работы алгоритма поиска может быть нахождение этого данного или отсутствие данного в таблице. В случае отсутствия данного возможны две операции:

1. Индикация того, что данного нет.
2. Вставка данного в таблицу.

Пусть *K* - массив ключей, тогда для всех *K(i)* существует *R(i)* - данное. *KEY* - аргумент поиска. Ему соответствует информационная запись *REC*. В зависимости от того, какова структура данных в таблице, различают несколько видов поиска.

Переупорядочение таблицы поиска.

Всегда можно говорить о некотором значении вероятности нахождения того или иного элемента.

P(i) - вероятность нахождения элемента.

В этом случае таблица поиска представляется как система с дискретными состояниями, а искомый элемент характеризуется *i*-ым состоянием системы, вероятность которого *P(i)*.

$P(1) + P(2) + ... + P(n) = 1$

Количество сравнений *Z* при поиске в таблице, т.е. количество сравнений, необходимых для поиска по заданному аргументу, представляет собой значение дискретной случайной величины, определяемой номерами состояний и вероятностями состояний системы.

$Z=1*P(1) + 2*P(2) + 3*P(3) + ... + n*P(n)$

Таблица данных должна быть упорядочена таким образом, чтобы в начале таблицы были элементы с большими вероятностями поиска или элементы, к которым чаще всего обращаются в таблице.

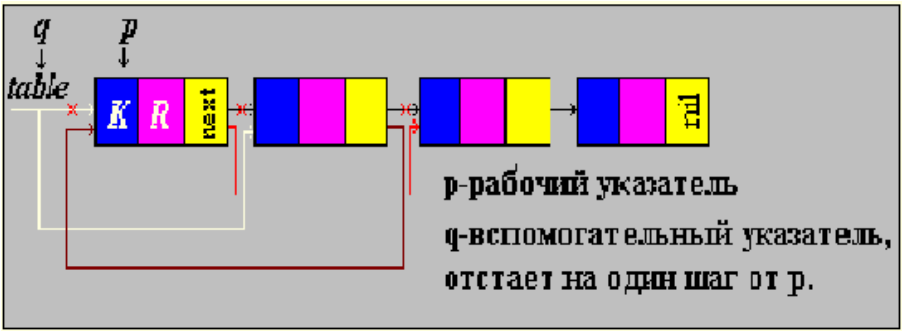
$P(1) >= P(2) >= P(3) >= ... >= P(n)$

Имеется два основных метода переупорядочивания таблиц поиска:

1) Переупорядочивание путем перестановки найденного элемента в начало списка.

2) Метод транспозиции.

Переупорядочивание путем перестановки найденного элемента в начало списка.



Найденный элемент сразу оказывается в голове списка.

На рисунке проиллюстрирован случай, когда найденный элемент второй в списке. Первоначально указатель *q* нулевой, указатель *p* указывает на начало списка; *p* перемещается на второй элемент, а *q* на первый. Указатель начала списка (*table*) перемещается на второй элемент, а указатель второго элемента на третий. Таким образом, второй элемент перемещается на первое место.

Метод транспозиции

В данном методе найденный элемент переставляется на один элемент к голове списка. Если к этому элементу обращаются часто, то он, постепенно перемещаясь к началу списка, скоро окажется в его голове.

Этот метод удобен тем, что он эффективен не только в списковых структурах, но и в неупорядоченных массивах, т.к. меняются местами только два рядом стоящих элемента.

Будем использовать три указателя:

p - рабочий указатель, указывает на элемент.

q - вспомогательный указатель, отстает на один шаг от *p*.

s - вспомогательный указатель, отстает на два шага от *p*.

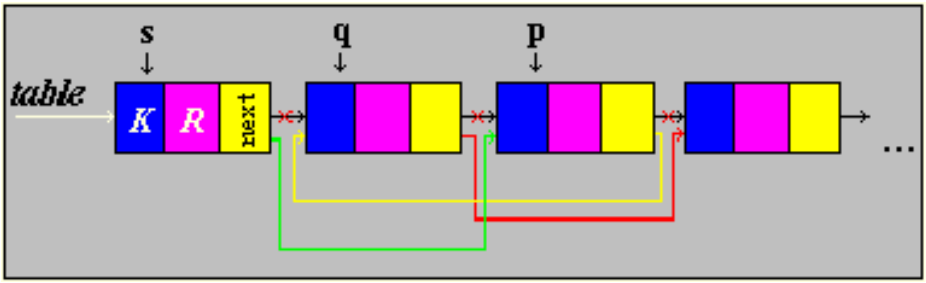


Рисунок иллюстрирует случай, когда найденный элемент третий в списке. Найденный нами третий элемент передвигается на один шаг к голове списка (т.е. становится вторым). Указатель первого элемента перемещается на третий элемент, указатель второго элемента на четвертый, таким образом, третий элемент перемещается на второе место. Если к данному элементу обратиться еще раз, то он окажется в голове списка.

5.3. Задания

Дан массив целых чисел. Решить заданную согласно варианту задачу и переставить найденный элемент обоими методами оптимизации поиска в начало списка.

1. Найти максимальный элемент массива.
2. Найти минимальный элемент массива.
3. Найти значение lg (x) от каждого элемента и переставить на 1 место элемент, значение функции от которого максимально.
4. Найти число, нацело делящееся на 11 (если таких чисел несколько, то найти максимальное; если таких чисел нет - выдать сообщение).
5. Найти элемент, разность соседних элементов которого не меньше 72. Если таких элементов несколько, выбрать максимальный. Если таких элементов нет, выдать сообщение.
6. Найти элемент, частное соседних элементов которого четное число. Если таких элементов несколько, выбрать максимальный или минимальный элемент. Если таких элементов нет, выдать сообщение.
7. Найти элемент, разность соседних элементов которого четное число. Если таких элементов несколько, выбрать максимальный или минимальный элемент. Если такого элемента нет, выдать сообщение.
8. Найти элемент, среднее арифметическое элементов,

находящихся до этого элемента равно 12. Если таких элементов нет, выдать сообщение.

9. Найти максимальный элемент, делящийся на 10. Если такого элемента нет, выдать сообщение.

10. Найти элемент, разность соседних элементов которого четное число и делится на 3. Если такого элемента нет, выдать сообщение.

11. Найти элемент, для среднее квадратичное элементов, находящихся после этого элемента меньше 10. Если таких элементов несколько, выбрать максимальный элемент. Если таких элементов нет, выдать сообщение.

12. Найти значение tg (x) от каждого элемента и переставить на 1 место элемент, значение функции от которого максимально.