**СТРУКТУРЫ И АЛГОРИТМЫ ОБРАБОТКИ ДАННЫХ**

**Литература**

1. Хусаинов Б.С. Структуры и алгоритмы обработки данных. Примеры на языке Си. М.: Финансы и статистика. 2004.

2. Кнут Д. Искусство программирования. В 3 томах. М.: Издат. Дом «Вильямс».2001.

3. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: Построение и анализ, 2-е изд. М.: Издат. Дом «Вильямс».2005.

Алгоритмы

*Алгоритмом* решения задачи называют конечную последовательность действий, которые надо выполнить над входными данными и промежуточными результатами для получения результата решения задачи.

Алгоритм должен обладать следующими свойствами:

1. Конечность – работа алгоритма должна заканчиваться за конечное число шагов, причём некоторые из шагов могут повторяться многократно.

2. Определённость (детерминированность) – все шаги алгоритма должны иметь однозначную трактовку и должны быть понятны исполнителю алгоритма, при одних и тех же исходных данных должны быть получены одни и те же результаты.

3. Массовость – алгоритм должен быть направлен на решение целого класса задач, отличающихся исходными данными.

4. Результативность – алгоритм должен приводить к достижению поставленной цели за конечное число шагов.

5. Эффективность – все шаги алгоритма должны выполняться за конечное время, а общее время решения задачи должно лежать в допустимых пределах.

*Основными этапами подготовки задачи к решению на компьютере являются*:

1. Постановка задачи – на этом этапе необходимо точно сформулировать и понять задачу, установить, имеет ли она решение; нельзя ли её решить с помощью готовых прикладных программ.

2. Построение модели задачи – предполагает подбор или разработку формального описания (модели) задачи, привлекая достижения современной науки.

3. Разработка алгоритма – в рамках построенной модели задачи разрабатываются или подбираются методы и алгоритмы её решения. При этом необходимо выяснить, существуют ли решения аналогичных задач, то есть руководствоваться накопленным опытом, что значительно сокращает сроки разработки, и, следовательно, снижает расходы.

4. Реализация алгоритма в виде программы – сначала выбирается язык программирования.

5. Проверка правильности алгоритма – выполняется на различных этапах. На этапе разработки алгоритма доказывается правильность каждого шага алгоритма, затем доказывается конечность алгоритма, проверяются допустимые входные данные и полученные по ним выходные данные. На этапе программной реализации необходимо следить за тем, чтобы в процессе преобразования правильного алгоритма получить правильную программу.

*Правильностью (корректностью)* программы называют свойство программы, характеризующее отсутствие в ней ошибок по отношению к целям разработки.

Каждая программа определяется правилами её представления (синтаксисом языка программирования) и правилами интерпретации её содержательного значения (семантикой). Корректная программа может быть создана только при строгом соотношении этих правил. Задача определения синтаксической правильности решается транслятором автоматически в ходе трансляции программы. Проблема доказательства семантической правильности оказывается значительно сложнее, так как процедура доказательства существенно зависит от решаемой задачи и исходных данных. Формализованное описание поставленной задачи называют *спецификацией* задачи. Область науки, занимающаяся доказательством соответствия между программной реализацией и спецификацией задачи, получила название *верификации* программ. Цель верификации программ – демонстрация их корректности.

Семантическая правильность программы проверяется путём её тестирования, то есть выполнения программы при определённых исходных данных (тестах) с целью нахождения ошибок.

6. Анализ сложности алгоритма: алгоритму решения задачи предъявляют два противоречивых требования:

а) быть простым для понимания, отладки и сопровождения;

б) быть эффективным в смысле затрат времени и памяти компьютера.

Какому из этих требований отдать предпочтение, определяется характером решаемой задачи.

При анализе с задачей связываются некоторое число, называемое *размером задачи* n, которое выражает меру количества (объёма) исходных данных. В качестве размера могут выступать размеры массивов, число вершин или дуг графа, число записей в файле и т.п. Время выполнения программы T зависит от n. Традиционно определяют время выполнения T(n) в наихудшем случае как меру сложности алгоритма – это так называемая *временная сложность*.

Типы данных

Данные характеризуются тремя качествами: семантикой, синтаксисом и прагматикой.

*Семантика* – это содержательная (смысловая) характеристика данных. Она определяет множество возможных значений данных и множество операций, которые можно выполнять над этими данными.

*Синтаксис* – это формальная характеристика данных, определяющая внешний вид (структуру) данных. Синтаксис основан на стандартах языка программирования и задаётся грамматикой, которая включает алфавит и синтаксические правила изображения данных.

*Прагматика* – характеристика, определяющая правила использования данных в компьютере в результате интерпретации данных, то есть прагматика данных – это алгоритмы их обработки в компьютере.

Комбинацию семантической и синтаксической характеристик данных в программировании называют *типом данных*. В современных языках программирования типы данных делятся на простые (скалярные) и структурированные.

Данные простых (скалярных) типов рассматриваются как цельные простые неделимые объекты. Например: целые, вещественные, символьный, логический и другие.

Данные структурированных типов представляются как составные агрегатные объекты, элементы которых могут быть данными одного и того же типа (в этом случае этот тип называется базовым) или различных типов.

Число различных значений, входящих в тип, называется мощностью этого типа.

Если для значений некоторого типа существует отношение порядка следования, то такой тип называется порядковым.

Абстрактный тип данных (АТД) – это комбинация всех трёх характеристик данных: семантической, синтаксической и прагматической, то есть это математическая модель данных с совокупностью операций, определённых в рамках этой модели. АТД представляются с помощью конкретного языка программирования в виде определённых синтаксических конструкций.

I. АЛГОРИТМЫ ПОИСКА

Данная глава посвящена решению простой поисковой задачи: находить данные, хранящиеся в памяти компьютера с определённой идентификацией. Например, в вычислительной задаче найти f(x) по значению x и таблице значений функции f; в лингвистической – найти английский эквивалент данного русского слова.

Предположим, что в файле (таблице) хранится множество из N записей и необходимо определить положение соответствующей записи. Каждая запись содержит специальное поле, которое называется *ключом*, идентифицирующее данную запись. Все N ключей различны, так что каждый ключ однозначно определяет свою запись. Совокупность всех записей называется *таблицей*. В алгоритмах поиска присутствует так называемый аргумент поиска К, и задача заключается в отыскании записи, имеющей К своим ключом.

Существует две возможности окончания поиска: либо поиск оказался удачным, то есть позволил определить положение соответствующей записи, содержащей К; либо неудачным, то есть показал, что К не может быть найден ни в одной из записей. После неудачного поиска иногда желательно вставить в таблицу новую запись, содержащую К. Алгоритм, который делает это, называется поиском со вставкой.

1. Последовательный поиск

Последовательные поиск реализует следующую процедуру: начни с начала и продвигайся, пока не найдёшь нужный ключ; тогда остановись. Рассмотрим три алгоритма.

*Алгоритм S (простой последовательный поиск).*

Имеется таблица записей R1,R2,…,Rn (N≥1), снабжённых ключами K1,K2,…,Kn. Алгоритм предназначен для поиска записи с ключом К.

1. Установить начальное значение i=1;

2. Если Ki=K, то алгоритм завершён (идти на п. 5), запись Ri – искомая;

3. Перейти к следующей записи: i=i+1;

4. Если i≤N, то перейти на шаг 2, иначе – искомой записи в таблице нет;

5. Конец алгоритма.

Время работы алгоритма зависит от количества прогонов цикла. Пусть обращение к записям происходит с равной вероятностью . Тогда среднее число прогонов цикла

; СКО числа прогонов цикла ≈0,289N.

i=1

Ki = K

да

нет (i=N+1)

нет

да

Поиск прошёл удачно

i = i + 1

i ≤ N

Поиск прошёл неудачно

Для ускорения работы алгоритма применяется *алгоритм Q**(быстрый последовательный поиск)* – предполагается, что в конце стоит фиктивная запись RN+1, ключ которой равен аргументу поиска.

1. Установка i=1, KN+1=K;

2. Если Ki=K, то перейти к п.4, иначе

3. Перейти к следующей записи: i=i+1 и вернуться к п.2;

4. Если i≤N, то поиск прошёл удачно, в противном случае – неудачно (i=N+1);

5. Конец.

Ускорение работы алгоритма до 30% по сравнению с алгоритмом S и достигается за счёт удаления одного сравнения из цикла.

i=1

KN+1=K

Ki = K

да

нет (i=N+1)

нет

да

Поиск прошёл удачно

i = i + 1

i ≤ N

Поиск прошёл неудачно

*Алгоритм Т**(последовательный поиск в упорядоченной таблице)*.

i=1

K ≤ Ki

да

нет (K < Ki)

нет (K>Ki)

да

Поиск прошёл удачно

i = i + 1

K = Ki

Поиск прошёл неудачно

Имеется таблица записей R1,R2,…,Rn,, причём ключи упорядочены по возрастанию: K1<K2<…<KN. Предполагается, что в конце таблицы расположена фиктивная запись RN+1, ключ которой KN+1>>К. Это позволяет избежать зацикливания при K> KN.

Если значение К с равной вероятностью принимает все возможные значения, то в случае удачного поиска алгоритм Т не лучше алгоритма Q. Однако отсутствие нужной записи алгоритм Т позволяет обнаружить примерно в два раза быстрее.

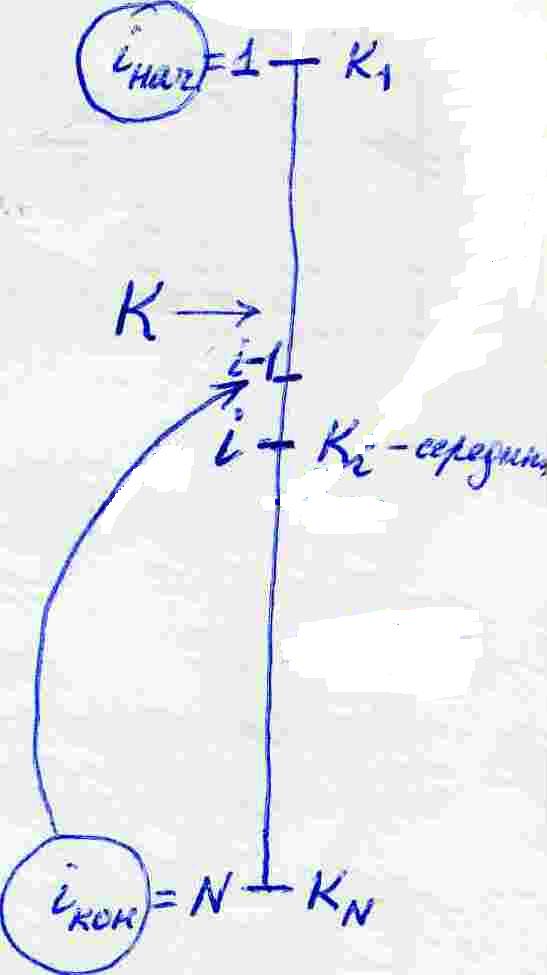
Приведенные выше алгоритмы предполагают отсутствие в таблице совпадающих значений ключей. В противном случае их необходимо модифицировать.

До сих пор предполагалось, что с равной вероятностью может потребоваться поиск любого аргумента, однако часто такое предположение неправомерно.

Пусть ключ Ki будет разыскиваться с вероятностью pi, причём p1+p2+…+pN=1. Время удачного поиска при больших N пропорционально числу сравнений С, среднее значение которого будет минимальна при p1≥p2≥…≥pN, то есть когда наиболее часто используемые записи расположены в начале таблицы.

Рассмотрим распределение

2. Прямой поиск в упорядоченной таблице

Снимем ограничение иметь лишь последовательный доступ к таблице, при этом получим возможность эффективно использовать отношение порядка в упорядоченной таблице.

Бинарный поиск (дихотомический, логарифмический)

Суть его: сначала сравнить К со средним ключом в таблице; по результату сравнения можно определить, в какой половине таблицы продолжить поиск, применяя при этом ту же процедуру. После не более чем log2N сравнений либо ключ будет найден, либо установлено его отсутствие.

*Алгоритм В (бинарный поиск)*. Найти аргумент К в таблице записей R1,R2,…,Rn, ключи которых упорядочены по возрастанию: K1<K2<…<KN. Пусть iнач и iкон – соответственно номера начальной и конечной записей в той области, в которой осуществляется поиск.

⌊·⌋ - целая часть (trunc).

Начальная установка:

iнач=1, iкон=N

iнач < iкон

да

нет (K>Ki)

да

Поиск прошёл удачно

Вычисление середины таблицы:

i = (iнач+iкон)/2

K < Ki

Поиск прошёл неудачно

iкон = i – 1

K = Ki

нет

да

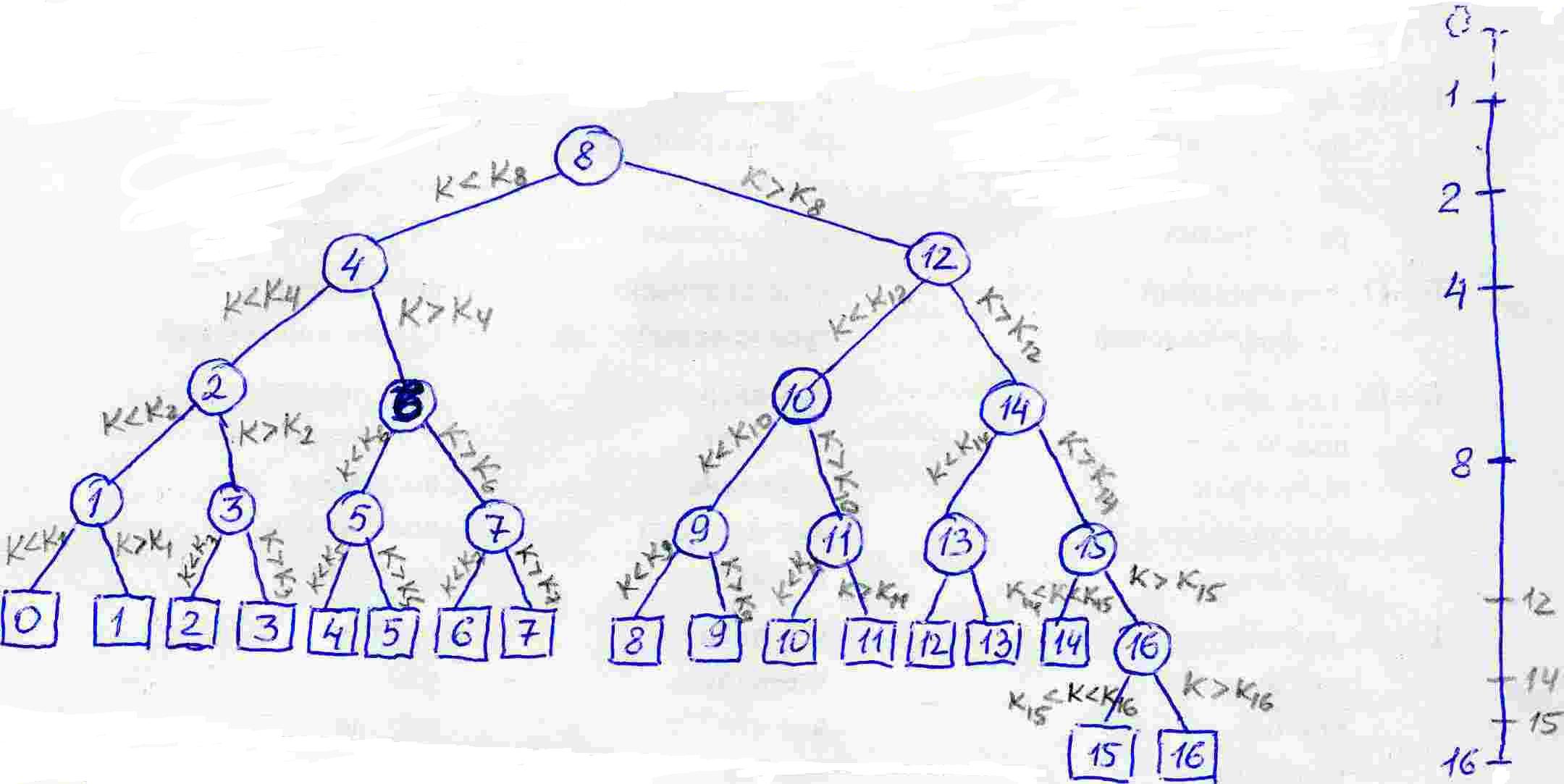
iнач = i + 1

нет

Представим бинарный поиск в виде бинарного дерева решений. Пусть N=16. Тогда «середина» , => К8 – ключ, соответствующий «середине» и первым (корневым) круглым узлом дерева будет 8.

Если К < К8, то поиск проводится в левом поддереве, сравнивая К с новой «серединой» К4 и т.д. Если К > К8, то поиск идёт в правом поддереве.

Неудачный поиск ведёт в один из внешних квадратных узлов от 0 до N. Например, узел 6 будет достигнут тогда и только тогда, когда К6 < К < К7.



С помощью дерева можно доказать, что справедлива теорема:

При 2k-1≤N≤2k удачный поиск, использующий алгоритм В, требует минимум 1, максимум К сравнений. Неудачный поиск требует k сравнений при N=2k-1 либо k-1 или k сравнений при 2k-1 ≤ N <2k-1.

Следствие: алгоритм В требует максимум [log2N]+1 сравнений, среднее число сравнений при удачном поиске .

II. СТРУКТУРЫ ДАННЫХ И СТРУКТУРЫ ХРАНЕНИЯ ДАННЫХ

Структуры данных

*Структурой данных* называют совокупность элементов данных, между которыми существуют некоторые отношения, причём элементами данных могут быть как один элемент (атомарные данные), так и структуры. Все связи одного элемента данных с другими образуют элемент отношений, связанных с соответствующим элементом данных. Пару, содержащую элемент данных и элемент отношений, назовём элементом структуры.



Рис. 1. Структура данных.

На основе фундаментальных структур в программе могут строиться более сложные структуры.

Таким образом, структура данных – это организованная некоторым образом информация, которая может быть описана, создана и обработана программами.

Структура данных используется для представления абстрактного типа данных.

Существуют линейные и нелинейные структуры данных. В линейных структурах между элементами имеется отношение соседства. К линейным структурам относятся массивы, очереди, стеки, деки, линейные списки. К нелинейным структурам данных относятся графы, деревья и леса.

Структуры хранения данных

Представление определённой структуры данных в памяти компьютера называется *структурой хранения данных*. Одной и той же структуре данных могут соответствовать различные структуры хранения, и, наоборот, разные структуры данных могут быть представлены одной и той же структурой хранения. Структура хранения, выбранная для представления конкретной структуры данных, должна, с одной стороны, сохранить отношения, существующие между элементами данных, с другой стороны, обеспечивать наиболее простое и эффективное выполнение операций над структурой данных в целом и отдельными её элементами. Такими операциями являются создание и уничтожение структуры данных, включение и исключение её элементов, доступ к элементам и т.п.

Большинство современных вычислительных машин имеет линейную оперативную память, состоящую из последовательности ячеек памяти. Ячейка – это наименьшая адресуемая часть памяти. Возможные значения адресов образуют адресное пространство модели вычислительной машины (0,1,2,…,n). По содержимому ячейки невозможно определить, какого типа данные в ней хранятся.

Для представления адреса в языках программирования имеется особый тип данных, называемый *указателем*. Возможными значениями данных типа «указатель» является всё адресное пространство компьютера. Указатель содержит адрес другого объекта программы, может содержать также адрес любой ячейки памяти компьютера. Если переменная содержит адрес некоторого другого объекта, то говорят, что переменная указывает (ссылается) на этот объект. Указатель может ссылаться на любой объект программы: простую переменную, массив, строку, структуру, функцию, другой указатель.

Рассмотрим возможности представления структур данных в линейной оперативной памяти.

Первая возможность связана с получением памяти. Данные могут размещаться либо в *статической*, либо в *динамической памяти*. Память под данные в программе может быть получена двумя способами. Первый способ – это когда для отдельных данных, которые описаны в программе, при её выполнении выделяется память, определяемая типом данных. Такая память является статической и сохраняется на время выполнения программы или её частей. (Не путать со статическими данными в языке Си и других языках программирования.) Второй способ – это получение памяти по явному запросу в программе. В запросе указывается объём требуемой памяти. В случае наличия памяти операционная система выделяет одну сплошную область памяти и возвращает адрес начала этой области. Такую память будем называть динамической памятью. За её правильное использование отвечает программист. После того как потребность в динамической памяти отпала, она может быть освобождена также по явному запросу. Динамическая память может быть получена для структуры данных в целом по одному запросу. Тогда размеры структуры данных могут изменяться только в пределах фиксированной длины полученной памяти, а число элементов в структуре будет ограничено. Если же получать память для каждого элемента по отдельному запросу, число элементов не ограничивается, пока есть доступная память вычислительной машины.

Вторая возможность определяется тем, является ли обязательным совместное размещение всех элементов многоэлементной структуры в одной сплошной области памяти либо допускается размещение отдельных элементов в различных областях памяти.

Третья возможность возникает, когда отдельные элементы многоэлементной структуры данных не имеют явных связей между собой и размещаются в одной сплошной области памяти последовательно, друг за другом.

Четвёртая возможность учитывается в том случае, когда между отдельными элементами многоэлементной структуры данных существуют явные связи по размещению их в памяти, и элементы структуры размещаются либо в одной сплошной области, либо в различных областях.

Используя различные допустимые сочетания этих возможностей, можно образовать различные структуры хранения данных.

Рассмотрим три вида структур хранения данных: вектор, список и сеть, на основе которых строятся многие другие структуры хранения.

1. Вектор

Вектор является наиболее простой структурой хранения многоэлементных структур данных. Он размещается в одной сплошной области памяти. Элементы структуры данных в векторе физически размещаются последовательно один за другим. Поэтому в элементе структуры данных для вектора имеется только элемент данных и отсутствует элемент отношений, показывающий связи элемента структуры данных с другими элементами.

Такая структура хранения наиболее близко соответствует реальной организации памяти большинства ЭВМ. Все элементы структуры данных имеют одинаковую длину. Доступ к отдельному элементу осуществляется с использованием базового адреса (адреса начала) вектора и номера этого элемента. Обычно в векторной памяти размещаются массивы (одномерные и многомерные), строки, стеки, очереди, таблицы и т.п.

Для манипулирования со структурой данных, хранимой в векторе, требуется управляющая информация, определяющая адрес начала вектора, тип элементов структуры данных и их количество в векторе и т.п. Для хранения управляющей информации создаётся *дескриптор* (описатель) вектора или информационный вектор.

Таким образом, структура вектора состоит из двух частей: дескриптора и непосредственно вектора, в котором последовательно размещаются элементы структуры данных (рис. 2).

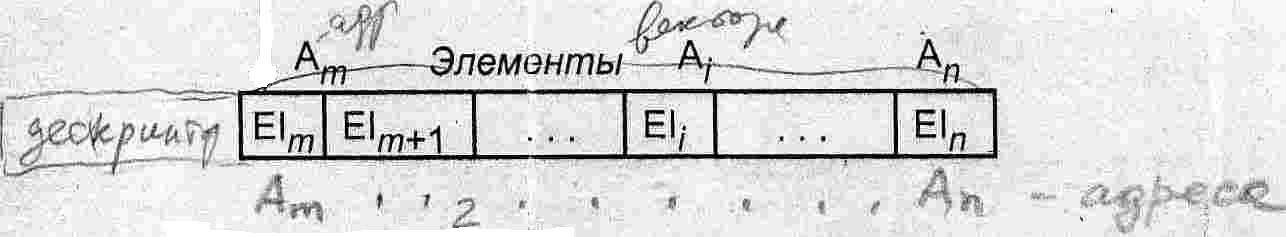


Рис. 2. Структура вектора.

Дескриптор вектора содержит: адрес Am начала вектора, тип элементов структуры данных или их длину L, число элементов или их начальный m и конечный n индексы. Адрес Ai элемента Ei с индексом i определяется выражением Ai = Am + (i - m) \* L.

В большинстве случаев отображение структур данных в структуры хранения в виде вектора можно выполнить, используя такой тип данных, как «массив». При этом транслятор создаёт как дескриптор вектора, так и сам вектор, а обращение к элементам вектора сводится к обращению к элементам массива. Однако транслятор создаёт статический массив с фиксированными размерами, что может явиться ограничивающим фактором его применения.

В прикладной программе можно организовать хранение данных в динамическом векторе. Для этого нужно выполнить ряд операций: 1) определить структуру дескриптора; 2) получить динамическую память под дескриптор и заполнить дескриптор управляющей информацией; 3) используя данные из дескриптора, определить размер памяти для вектора и получить динамическую память; 4) при необходимости инициализировать элементы структуры данных в векторе. Адреса элементов к ним вычисляются с использованием управляющей информации из дескриптора. В программе доступ к структуре данных, хранимой в виде вектора, осуществляется посредством указателя на дескриптор (рис. 3).



Рис. 3. Доступ к вектору через дескриптор.

Иногда вместо дескриптора используется только указатель на начало вектора. Тогда остальные параметры структуры данных (тип или длина элементов, число элементов и т.п.) должны быть известны каждой из процедур обработки структуры данных, что может усложнить их реализацию.

2. Список

Список является более сложной структурой хранения, в которой элементы имеют явные связи. В списке элементы структур данных размещаются не обязательно в смежных областях памяти. Каждый элемент структуры данных содержит, по крайней мере, два поля. Одно поле – основное содержит данные этого элемента или указатель на данные, а другое поле предназначено для размещения указателя на следующий элемент. Оконечный элемент структуры данных в поле указателя содержит признак конца списка (NULL – в Си, в Паскале - Nil). Элемент структуры данных, в свою очередь, тоже может быть структурой, хранимой в виде списка. Различают списки *однонаправленные, двунаправленные и циклические*. Число элементов списка может изменяться. Учитывая наличие явных связей между элементами списка, иногда используют термины цепной список или связной список. Список, отражающий отношение соседства между элементами (у каждого элемента списка могут быть только один предыдущий и один последующий элементы), называется *линейным*. Любой другой список называется нелинейным.

Однонаправленный список (рис. 4) является простейшей формой списка.



Рис. 4. Однонаправленный список.

Достоинства однонаправленного списка – простота создания, включения и удаления элементов. Недостаток – просмотр элементов осуществляется всегда с начала списка. Правда, программа обработки может запоминать указатель на текущий элемент, тогда обработку можно продолжать с него.

Циклический однонаправленный список (рис. 5) отличается от однонаправленного только тем, что последний элемент в поле указателя содержит не признак конца списка, а ссылку на начало списка, т.е. на первый элемент, что позволяет перейти на другую операцию обработки элементов, начиная с любого элемента списка.



Рис. 5. Циклический список.

В двунаправленном списке каждый элемент содержит два указателя – на последующий и предыдущий элементы, что позволяет вести обработку элементов как в прямом, так и в обратном направлении. Двунаправленный список также может быть циклическим.

Физически список реализуется как совокупность дескриптора списка и элементов, размещённых в оперативной памяти или на внешних запоминающих устройствах (ВЗУ) и связанных друг с другом в цепочку с помощь указателей. Дескриптор обычно реализуется в виде записи и может содержать следующую информацию о списке:

* тип структуры;
* имя списка;
* указатель (адрес) начала списка;
* текущее число элементов;
* описание элемента (длина, тип и т.п.).

Вместо дескриптора можно использовать только указатель.

Элементы структуры данных в списке могут быть как одинаковых, так и различных типов. Обработка элементов в первом случае проще, однако, иногда это приводит к излишним затратам памяти. Если данные существенно различаются по длине. Если данные различаются только по длине, то каждый элемент должен содержать информацию о своей длине. При различии по формату элементы снабжаются дескрипторами, определяющими формат и длину. Однако в этом случае выгоды от экономии памяти могут быть потеряны за счёт усложнения процедур обработки элементов.

Элементы структуры данных в списке могут быть упорядочены по возрастанию или убыванию значения некоторого поля данных (по ключу). Обработка упорядоченных списков во многих случаях проще обработки неупорядоченных списков.

Списки позволяют свободное размещение отдельных элементов структуры данных в различных областях динамической памяти, что обеспечивает более гибкое использование памяти ЭВМ. Тогда для включения в список нового элемента запрашивается динамическая память, а при удалении элемента из списка занятая им память освобождается. Это обеспечивает возможность гибкого использования памяти, но получение и освобождение памяти сопряжены с обращением к средствам управления памятью операционной системы и приводит к дополнительным затратам времени.

Возможна организация списка и в смежной области памяти. В этом случае все элементы структуры данных имеют один и тот же тип, а число возможных элементов ограничено размерами области памяти. На рис. 6 приведён пример цепного списка, размещённого в одной сплошной области памяти (указатель списка размещён в другой области и содержит адрес списка). Элементами данных являются буквы. Последовательность букв, размещённых в списке, образует слово АЗБУКА.



Рис. 6. Список в смежной области памяти.

Если обработка списка предусматривает операции как включения, так и удаления элементов, то потребуются создание и ведение двух дополнительных списков: списка свободных и списка занятых элементов. Удаляемый элемент при этом включается в список свободных, а для включения элемента используется элемент из списка свободных, если таковые имеются. В качестве указателей в таких списках могут использоваться либо индексы элементов, либо их адреса.

3. Сеть

Сеть как структура хранения является дальнейшим развитием списковой структуры хранения. Каждый элемент структуры данных в сети, кроме данных, может содержать несколько указателей на другие элементы. На один и тот же элемент могут ссылаться несколько других элементов, образуя, таким образом, сложные переплетения по связи (рис. 7).

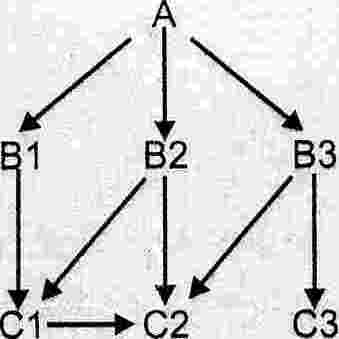


Рис. 7. Сеть.

В общем случае количество полей в элементах структуры данных в сети может быть различным, тогда элемент должен содержать информацию о своём формате. Вместо данных в элементе сети, как в списке, может располагаться ссылка на данные.

Сети используются для представления в памяти ЭВМ сложных структур данных, таких как графы, деревья, многоуровневые и многосвязные таблицы, структуры сложных программных изделий, организационные структуры предприятий и связи между ними и т.п.

Каждый элемент структуры данных в сети обычно размещается в динамически получаемой памяти, что обеспечивает гибкость представления и обработки данных. Сети могут быть организованы и в смежной области памяти фиксированной длины, если количество элементов и структура данных, представленная сетью, при решении задачи остаются неизменными.

*Далее приводится материал, непосредственно не относящийся к «Структурам и алгоритмам обработки данных», но необходимый для выполнения лаб.раб.№2, 3, касающийся динамической памяти и указателей*

Динамическая память

Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти, которая называется сегментом данных. Длина сегмента данных определяется архитектурой микропроцессоров 80х86 и составляет 65536 байт, что может вызвать затруднения при обработке больших массивов данных. Выходом из положения может служить использование так называемой динамической памяти.

Динамическая память – это оперативная память компьютера, предоставляемая программе при её работе, за вычетом сегмента данных (64 Кбайт), стека (обычно 16Кбайт) и собственно тела программы. Размер динамической памяти можно варьировать в широких пределах. Динамическая память – это фактически единственная возможность обработки массивов данных большой размерности. Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. В отличие от этого статическое размещение осуществляется компилятором Турбо Паскаля в процессе компиляции программы. При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных, к ним нельзя обращаться по именам, как к статическим переменным.

Адреса и указатели

Оперативная память компьютера представляет собой совокупность элементарных ячеек для хранения информации – байтов, каждый из которых имеет собственный номер (адрес). Турбо Паскаль предоставляет в распоряжение программиста гибкое средство управления динамической памятью – так называемые указатели. Указатель – это переменная, которая в качестве своего значения содержит адрес байта памяти. В персональных компьютерах адреса задаются совокупностью двух 16-разрядных слов, которые называются сегментом и смещением. Сегмент – это участок памяти, имеющий длину 64 Кбайт и начинающийся с физического адреса, кратного 16 (0, 16, 32, 48, …). Смещение указывает, сколько байт от начала сегмента необходимо пропустить, чтобы обратиться к нужному адресу. Таким образом, по своей структуре любой указатель представляет собой совокупность двух слов (данных типа word), трактуемых как сегмент и смещение.

Объявление указателей

Как правило, в Турбо Паскале указатель связывается с некоторым типом данных. Такие указатели будем называть типизированными. Для объявления типизированного указателя используется значок ^, который помещается перед соответствующим типом.

Var p1:^integer; p2:^real;

Type

PerconPointer = ^PerconRecord;

PerconRecord = record

Name: string;

Job: string;

Next: PerconPointer

end;

При объявлении типа PerconPointer дана ссылка на тип PerconRecord, который описан уже позже. Это исключение сделано для указателей, которые могут ссылаться на ещё не объявленный элемент данных.

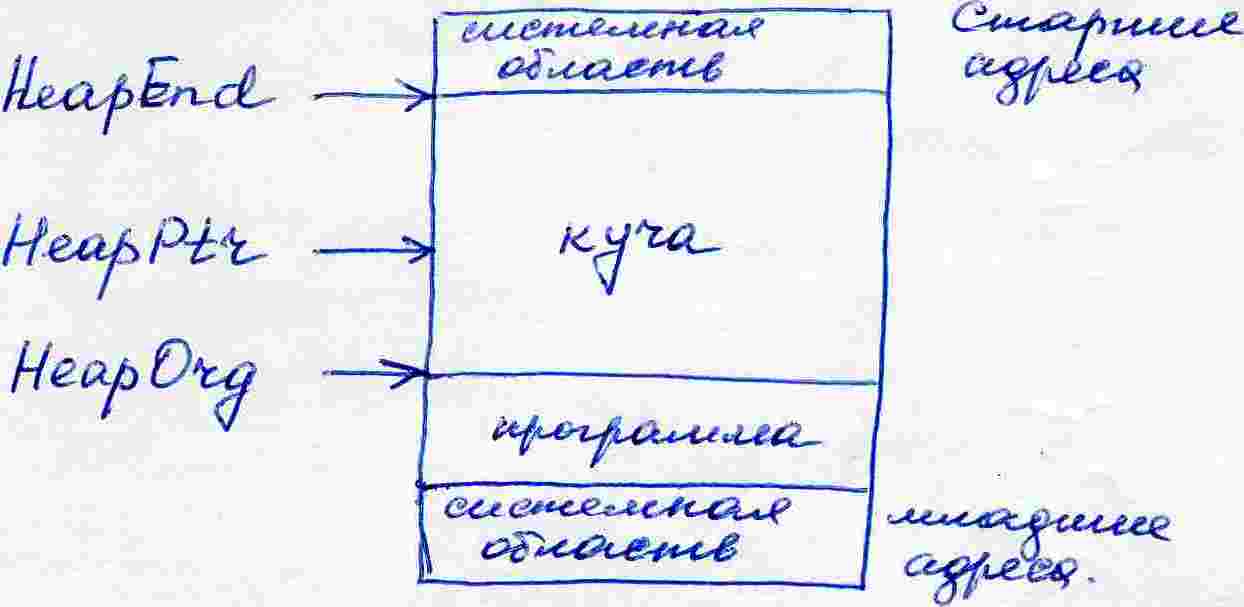
В Турбо Паскале можно объявлять указатель и не связывать его при этом с каким-либо конкретным типом данных. Для этого служит стандартный тип pointer: Var p:pointer;

Указатели такого рода называются нетипизированными. Поскольку они не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

В Турбо Паскале можно передавать значения только между указателями, связанными с одним и тем же типом данных. Например, если Var p1,p2:^integer; p3:^real; pp:pointer; то присваивание p1:=p2; допустимо, а p1:=p3; - нет, так как р1 и р2 указывают на разные типы данных. Это ограничение не распространяется на нетипизированные указатели, по этому можно записать рр:=р3; р1:=рр; и тем самым достичь нужного результата.

Выделение и освобождение динамической памяти

Вся динамическая память в Турбо Паскале рассматривается как сплошной массив байтов, который называется кучей. Физически куча располагается в старших адресах сразу за областью памяти, которую занимает тело программы. Начало кучи хранится в стандартной переменной heaporg, конец – в переменной heapend. Текущую границу занятой динамической памяти указывает указатель heapptr.



Память под любую динамически размещаемую переменную выделяется процедурой New. Параметром обращения к этой процедуре является типизированный указатель. В результате обращения указатель приобретает значение, соответствующее динамическому адресу, начиная с которого можно разместить данные. Например:

Var i,j:^integer; r:^real;

Begin

New(i);

…

end.

После выполнения этого фрагмента указатель i приобретает значение, которое перед этим имел указатель кучи HeapPtr, а сам HeapPtr увеличит своё значение на 2, так как длина внутреннего представления типа integer, с которым связан i, составляет 2 байта. Оператор New(r); вызовет ещё раз смещение HeapPtr на 6 байт (длина внутреннего представления типа Real = 6 байт).

После того как указатель приобрёл некоторое значение, то есть стал указывать на конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа. Для этого сразу за указателем без пробела ставится ^, например:

i^:=2; {в область памяти i помещено значение 2}

r^:=2\*pi;

Таким образом, значение, на которое указывает указатель, то есть собственно данные, размещённые в куче, обозначаются значком ^, который ставится сразу за указателем. Если за указателем нет значка ^, то имеется в виду адрес, по которому размещены данные. То есть значением любого указателя является адрес, а чтобы указать, что речь идёт не об адресе, а о тех данных, которые размещены по этому адресу, за указателем ставится ^. Поэтому допустим оператор r^:=sqr(r^)+i^-1; где r^ - значение, и недопустим r=sqr(r^)+i^-1; где r - адрес, ему нельзя присвоить значение вещественного выражения.

Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура dispose. Например, оператор dispose(i); вернёт в кучу 2 байта, а dispose(r); - 6 байт, которые раньше были выделены указателям i и r.

Отметим, что процедура dispose(ptr); не изменяет значения указателя ptr, а лишь возвращает в кучу память, ранее связанную с этим указателем. Повторное применение процедуры dispose к уже свободному указателю приведёт к ошибке. Поэтому освободившийся указатель программист может пометить зарезервированным словом Nil, тогда можно будет организовать проверку, свободен этот указатель или нет, например:

const p:^real = Nil;

begin

…

if p=nil then New(p);

…

dispose(p);

p:=Nil;

…

end.

Другие операции, кроме присваивания и сравнения, над указателями не разрешены.

Существует возможность освобождения целого фрагмента кучи. С этой целью перед началом выделения динамической памяти текущее значение указателя HeapPtr запоминается в переменной-указателе с помощью процедуры MARK. Теперь можно в любой момент освободить фрагмент кучи, начиная от того адреса, который запомнила процедура MARK, и до конца динамической памяти. Для этого используется процедура release. Пример:

Var p,p1,p2,p3,p4,p5:^integer;

Begin

new(p1); new(p2); mark(p);

new(p3); new(p4); new(p5);

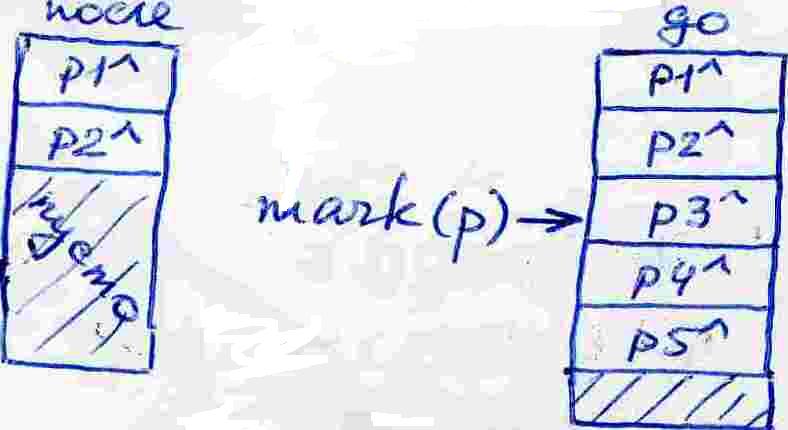
…

release(p)

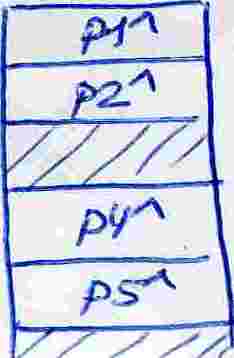
end.

В указатель р процедурой mark(p) было помещено текущее значение HeapPtr, однако память под переменную р не резервировалась. Обращение release(p) освободило динамическую память от помеченного места до конца кучи.

Состояние динамической памяти до и после работы этой программы приведено на рисунке.



Если заменить release(p) на, например, dispose(p3), то состояние памяти будет такое («ячеистая структура»).



Вызов release уничтожает список свободных фрагментов в куче, созданных до этого процедурой dispose, поэтому совместное использование обоих способов освобождения памяти (new – dispose и new – mark - release) в одной программе использовать не рекомендуется.