Лекция 1. Базовые понятия и методы хранения сложных структур данных

Программа = Данные + Алгоритм

Разработка программы включает параллельное проектирование этих двух ее компонентов. На каждом этапе проектирования сначала уточняется структура данных, а затем алгоритм, т.е. операции над данными.

Данные - это информация, представленная в форме, воспринимаемой компьютером.

Используемые далее в тексте обозначения видов данных:

Обозначение	**Смысл**
m..n	Последовательность целых чисел m, m+1, …, n
X[m..n]	Отрезок массива X[m], X[m+1], ..., X[n].
Очередь <== x;	Включить в Очередь значение x
Очередь ==> x;	Исключить из Очереди элемент и записать в x.
Очередь ==> ;	Исключить из Очереди элемент (без запоминания)
Стек <== x;	Втолкнуть в Стек значение x
Стек ==> x;	Вытолкнуть из Стека элемент и записать в x
Стек ==> ;	Вытолкнуть из Стека элемент (без запоминания)
X ↔ Y	Обменять местами значения X и Y
for (X ∈ S)	Повторять цикл для X, равного каждому элементу множества S (в произвольном порядке)
(условие)	1, если условие (например, X > 0) - истинно, 0 в противном случае

1.1. Уровни описания данных

Часто, например, используют уровня описания структуры данных: функциональный, логический и физический.

1. На *функциональном уровне* определяются необходимые для решения задачи операции над структурой данных и их свойства. Выбирается информационная структура данных для решаемой задачи.

2. На *логическом уровне* эта информационная структура данных структура данных разбивается на отдельные элементы с учетом математической обработки.

3. На *физическом уровне* для выбранных элементом и математического аппарата их обработки определяется конкретный способ представления данных в компьютере и набор операций с ними в выбранном вами языке программирования. Достаточно выразить данные и операции решаемой задачи

через базовые структуры и понятия языка программирования, так как детальное представление операций на уровне компьютера (машинного языка) уже автоматически определяется транслятором. Программист же имеет дело с операциями представления и обработки данных на языке программирования высокого уровня.

На функциональном и логическом уровнях имеют дело с абстрактными (математическими) структурами данных, организованными в соответствии с решаемой задачей без детального учета особенностей обработки этих данных компьютером и средствами их описания на языке программирования.

Абстрактная структура данных - это структура данных, рассматриваемая с точки зрения применяемых к ней операций без описания способа ее представления в памяти. Она близка к понятию абстрактного типа данных.

Абстрактный тип данных - тип данных, определяемый функционально: только через операции над объектами этого типа без описания способа представления их значений.

На физическом уровне имеют дело *со структурами хранения*, т. е. структурами данных, реализованные с учетом физической организации данных в компьютере и легко реализуемыми в языке программирования. В качестве базовых структур хранения в учебнике рассматриваются *вектор, список и сеть.*

Множество абстрактных структур данных бесконечно, т. к. для каждой задачи могут изобретаться свои структуры. Наиболее распространенные в самых разных задачах абстрактные структуры данных таковы: *очередь, стек, дек, строка, граф, дерево, таблица, массив и множество*.

Каждая структура данных представляется по единому плану: определение, применение, типовые операции, способы представления данных и реализации операций.

Абстрактную структуру данных можно реализовать разными способами на базе других более простых структур данных. Основными критериями выбора метода реализации являются:

1) возможность реализации требуемых операций над данными;
2) время выполнения операций;
3) требуемая память;
4) простота программирования.

Относительная важность этих критериев зависит от конкретной ситуации. Критерии часто противоречат друг другу, и разработчику приходится искать разумный компромисс между ними.

1.2. Физические данные. Методы хранения данных

Возможные методы физического хранения данных в компьютере определяются организацией *оперативной памяти*.

В большинстве современных компьютеров оперативная память построена по *адресному принципу* и представляет собой пронумерованную последовательность ячеек одинакового размера. Номер ячейки называется ее *адресом*, содержимое ячейки - *машинным словом*. Количество ячеек (объем, емкость памяти) обычно находится в пределах от 10^3 до 10^{10}, а размер (длина) ячейки – от 8 до 64 бит.

Адресом данных, занимающих несколько соседних ячеек, считают адрес первой из них. В языках программирования адреса называют также *указателями* или *ссылками*.

Минимальным элементом хранения является *бит* - двоичная цифра, принимающая одно из двух значений, 0 или 1. Бит используется также как единица количества информации. Количество информации в битах равно минимальному числу двоичных цифр, необходимому для представления этой информации.

Машинное слово представляет собой команду или данные. Первоначально ЭВМ использовались преимущественно для обработки числовой информации. Ячейка памяти содержала одно число и имела длину от 16 до 64 бит. Это неудобно для представления символьной (текстовой) информации, т. к. код символа (*байт*) в зависимости от размера алфавита содержал от 5 до 8 бит, и в ячейке приходилось размещать несколько символов, что затрудняло доступ к каждому из них.

С повышением роли обработки символьной информации стали применять *байтовую организацию памяти*, когда ячейка содержит один байт, равный 8 бит, а для представления числа используются несколько соседних байтов. Байт также рассматривается как единица количества информации, равная одному символу текста.

Таким образом, оперативная память компьютера имеет линейную (одномерную) организацию, и для хранения многомерных массивов и других сложных структур данных их необходимо *"линеаризовать".*

Существуют три основные группы методов хранения структур данных.

1. ***Последовательное представление данных***. Элементы структуры располагаются в памяти друг за другом без промежутков. Наиболее используемой структурой хранения является **вектор**.

2. ***Связанное представление данных***. Элементы структуры могут размещаться в памяти в произвольном порядке не обязательно подряд, причем каждый элемент содержит указатели (адреса) одного или нескольких других элементов, позволяющие отыскивать их в памяти. Основные структуры хранения - **список и сеть**.

3. ***Адресная арифметика***. Элементы структуры располагаются в памяти в произвольном порядке с возможными промежутками, но существует

определенная *закономерность*, позволяющая *вычислять адрес элемента*, например, в зависимости от его номера или другого параметра. Адресная арифметика может рассматриваться как обобщение последовательного и связанного представлений, в общем виде используется сравнительно редко.

Вектор - это набор элементов одинакового размера, расположенных в памяти подряд. Под набором будем понимать *конечное множество*. Вектор определяется *базой* (адресом первого элемента), *длиной* (количеством элементов) и *размером элемента*. По сути дела, вектор представляет собой **одномерный массив**. Главное достоинство вектора - *прямой доступ* к элементу по его порядковому номеру - *индексу*:

$$адрес(V[J]) = адрес(V[0])+d*J = адрес(V[1])+d*(J-1), \qquad \textbf{(1.1)}$$

где V[J] - элемент с индексом J, d - размер элемента (количество ячеек, занимаемых одним элементом); для простоты считаем, что $d \geq 1$, целое.

Элементы вектора одинаково доступны и их можно обрабатывать в *любом порядке*.

Список (или *связанный список*) - это *последовательность* элементов, каждый из которых, кроме других данных, содержит *указатель* (адрес) следующего элемента. Графически указатель изображают в виде стрелки, соединяющей элементы списка. На рис. 1.1 показан список, каждый элемент которого содержит один символ. Последний элемент списка содержит *пустой указатель* ("адрес" несуществующего объекта), показанный в виде поля, перечеркнутого крестиком 'X'.

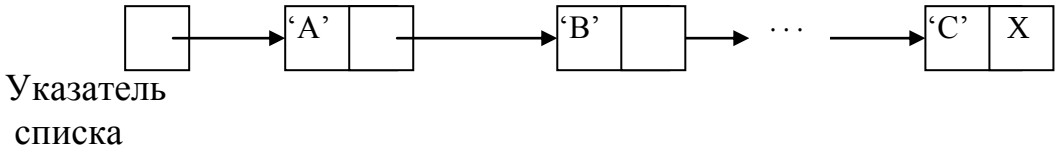

Рис. 1.1. Список

Сеть (многосвязный список) - это набор элементов, каждый из которых может иметь несколько указателей (ссылок) на другие элементы. В *однородной* сети все элементы содержат одинаковое количество ссылок, в *неоднородной* - разное (рис. 1.2).

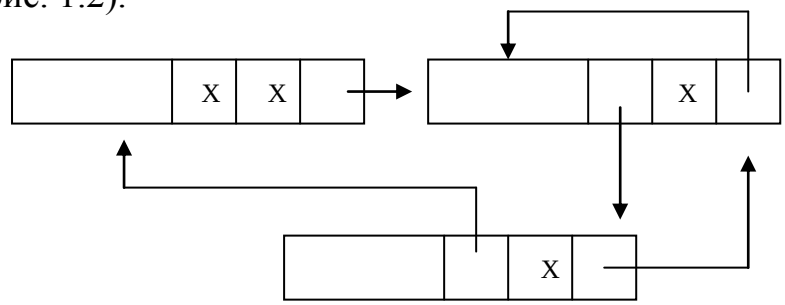

Рис. 1.2. Однородная (*регулярная*) сеть из трех элементов

Главное достоинство списков и сетей - простота добавления и удаления элементов; недостатки - *дополнительная память для указателей* и возможность только *последовательного доступа* к элементам (допускается движение только по ссылкам).

Наиболее распространенным средством представления структур данных в языках программирования являются *массивы*. Одномерный массив (*вектор*), имеется в любом языке высокого уровня и легко реализуется в компьютере на машинном языке. Вектор рассматривается как базовая структура хранения, а многомерный массив - как производная от вектора структура данных.

Современные языки включают понятие "*структура*" (C, PL/1 и др.) или "запись" (Pascal и др.). *Структура (запись)* - это совокупность поименованных элементов - полей - одного или разных типов. Это понятие очень близко к понятию структуры данных и позволяет его формализовать.

Отсутствие в языке записей до некоторой степени можно компенсировать *параллельными массивами*. Совокупность (массив) из n однотипных записей заменяется *параллельными массивами*, количество которых равно количеству полей в записи. Каждый массив содержит n значений соответствующего поля всех записей.

Понятие структуры данных близко к понятию типа данных в языке программирования. Абстрактные структуры данных называют еще абстрактными типами данных. Абстрактные типы данных лежат в основе методики *объектно-ориентированного программирования.* Их изучение служит, в частности, базой для освоения этой современной технологии.

В языке C++ для объектно-ориентированного программирования понятие структуры обобщается и приближается к концепции абстрактного типа данных: **элементами структуры могут быть не только данные, но и допустимые для них операции**. Такая структура в C++ называется *объектом*, тип объектов - *классом*.

Пример 1.1. Определение массива s из 100 записей (структур) с тремя полями sim, i, x:

```
struct
{ char sim;              /* Символьное поле sim                          */
  int i;                 /* Целочисленное поле i                         */
  float x[10];           /* Поле x - массив из 10 вещественных чисел     */
} s[100];
```

можно заменить определением трех параллельных массивов по 100 элементов:

```
char   sim[100];          /* Поля sim всех 100 записей                   */
```

```
int     i[100];          /* Поля i всех 100 записей          */
float   x[100][10];      /* Поля x всех 100 записей          */
```

Тогда, например, поле sim 15-й записи
s[15].sim заменится на sim[15],
5-й элемент поля x 20-й записи
s[20].x[5] запишется как x[20][5],

но вся 40-я запись s[40] не имеет аналогичного обозначения в параллельных массивах.

1.3. Анализ сложности алгоритмов

На всех этапах разработки программ необходимо анализировать их характеристики, к которым прежде всего относятся надежность и эффективность. Надежность программ тесно связана с их формальной корректностью.

На практике для анализа и обеспечения корректности проводят *верификацию* (доказательство правильности) и *тестирование* (проверочное исполнение) наиболее важных и сложных частей разрабатываемого алгоритма.

Верификация включает доказательство конечности процесса исполнения и правильности результата алгоритма. Обычно используется метод математической индукции.

В некоторых случаях для пояснения и обоснования алгоритма используется *инвариант цикла* - утверждение, соблюдаемое при каждом попадании управления в некоторую точку цикла. Формулирование инварианта цикла - наиболее трудная часть доказательства, проверить его правильность гораздо легче. Инвариант цикла выражает основную идею его построения и его удобно записывать в виде комментария в соответствующем месте цикла.

Для пояснения работы алгоритма мы будем использовать трассировочную таблицу его тестирования.

Эффективность программы определяется *временем* работы алгоритма и *объемом памяти* для алгоритма и данных. Важную роль в программировании играет *пространственно-временная закономерность*: уменьшение времени работы алгоритма приводит, как правило, к дополнительным затратам памяти и, наоборот, экономия памяти обычно приводит к замедлению работы программы. Таким образом, эти два параметра программы тесно взаимосвязаны. Изменяя один из них, можно в определенной степени управлять значением другого.

Под *сложностью* алгоритма имеют в виду не трудность его разработки или понимания, а его **вычислительную сложность**, т.е. время и объем памяти, необходимые для решения задачи. В примерах приводятся оценки сложности ряда алгоритмов и методов решения задач. Сложность алгоритма обычно

оценивается асимптотически - **по скорости роста необходимого времени и памяти при возрастании размера задачи.**

Размер задачи **n** определяется количеством обрабатываемых данных: длиной входной последовательности, размером массивов, числом вершин или ребер графа и т. д. Время измеряется количеством основных элементарных операций, необходимых для решения задачи (пересылка, сложение, сравнение и т. п.).

Для сравнения скорости роста функций в литературе используются различные обозначения.

O-обозначение, **Ω**-обозначение и **Θ**-обозначение

Выражение f(n) = **O(**g(n)**)** читается "f(n) есть о большое от g(n)" (говорят также “f(n) имеет значение порядка g(n)”) и обозначает, что g(n) является *верхней границей* для f(n), то есть f(n) ≤ C·g(n) для некоторой константы C.

Выражение f(n) = Ω(g(n)) читается "f(n) есть омега большое от g(n)" и обозначает, что g(n) может служить *нижней границей* для f(n).

Выражение f(n) = Θ(g(n)) читается "f(n) есть тета большое от g(n)" и обозначает, что g(n) является асимптотически *точной оценкой* для f(n).

Справедливы отношения:

f(n) = Ω(g(n)) <==> g(n) = O(f(n)),
f(n) = Θ(g(n)) ==> g(n) = Θ(f(n)),
f(n)=Θ(g(n)) <==> f(n)=O(g(n)) и g(n)=O(f(n)).

Например, $n^5+6*n = O(n^5)$, величина порядка O(1) не зависит от *n* (т. е. является константой).

Таким образом, утверждение, что время или память имеет значение порядка *f(n)* или *O(f(n))* означает, что при увеличении размера задачи *n* эта величина не превышает *Cf(n)*, где *C = const >0*.