Структуры данных

Введение

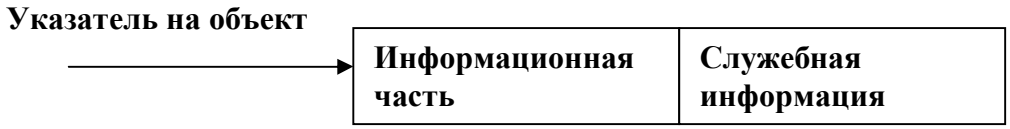
Известно, что все переменные, встречающиеся в программе, должны быть описаны. Перед началом выполнения программы каждой переменной для размещения её значения выделяется место в сегменте данных. Размер выделяемого места зависит от типа переменной. Например, для переменной типа Integer выделяется 2 байта. Обращение в программе к объекту, размещённому в некотором месте памяти, осуществляется с помощью имени переменной. Соответствие между переменной и сопоставленным ей местом в памяти сохраняется для описанных в программе переменных на всём протяжении выполнения программы. Имеются средства, позволяющие заниматься отведением и освобождением памяти для размещения объектов того или иного типа непосредственно по ходу выполнения программы. Память в этом случае отводится в динамической области, которая **называется областью кучи** или просто **кучей** (ещё она называется Heap-областью). **Данные,** размеры которых задаются непосредственно во время выполнения программы, называются **динамическими**. Для объявления динамических данных используется **ссылочный тип**, называемый также **типом-указателем**. С помощью ссылочного типа можно объявлять переменные, значением которых будет адрес ячейки памяти.

В программировании часто приходится иметь дело с множествами, меняющимися в процессе выполнения алгоритма. Для хранения таких массивов используются динамические структуры данных.

Разные алгоритмы используют разные операции. Нередко, например, требуется лишь добавлять и удалять элементы в множество, а также проверять, принадлежит ли множеству данный элемент. Структура данных,

поддерживающая такие операции, называется словарём. В других ситуациях могут понадобиться более сложные операции. Понятно, что выбор реализации динамического множества зависит от того, какие операции с ним нам потребуются.

Обычно элемент динамического множества – это запись, содержащая различные поля. Часто одно из полей рассматривается как **ключ**, предназначенный для идентификации элемента, а остальные поля – как дополнительные данные, хранящиеся вместе с ключом (эти поля также называют **информационной частью списка**). Элемент множества ищется по ключу; когда элемент найден, мы можем прочесть или изменить дополнительную информацию, имеющуюся в этом элементе.



Многие способы реализации множеств требуют, чтобы вместе с каждым ключом хранились не только дополнительные данные, но и некоторая служебная информация (например, указатели на другие элементы множества). Можно предположить, что элементы множества хранятся в некоторой общей области памяти и для каждого элемента имеется указатель, который позволяет получить доступ к этому элементу. Обычно в качестве указателя выступает просто адрес в памяти.

Если язык программирования этого не предусматривает, указателем может быть индекс в массиве.

Элементарные структуры данных

Операции над множествами

Главная возможность, которую предоставляет наличие ссылочных типов и ссылочных переменных – это возможность построения с их помощью объектов со сложной меняющейся структурой. Примерами таких структур являются множества, списки, стеки, деревья …

Операции над множествами делятся на запросы, не меняющие множества, и операции, меняющие множество. Типичные операции над множествами:

НАЙТИ(A,к) – запрос, возвращает указатель на элемент с ключом к в множестве A, если такой элемент не найден возвращается ПУСТО

ДОБАВИТЬ(A,x) – добавляет к множеству A элемент, на который указывает указатель x (подразумевается, что к этому моменту все поля в записи, на которую указывает x, уже заполнены).

УДАЛИТЬ(A,x) – удаляет из множества A элемент, на который указывает указатель x (обратите внимание, что x – указатель, а не ключ).

МИНИМУМ(A) – выдаёт указатель на минимальный ключ из множества.

МАКСИМУМ(A) - выдаёт указатель на максимальный ключ из множества.

СЛЕДУЮЩИЙ(A,x) – возвращает указатель на элемент, следующий за элементу с указателем x или ПУСТО.

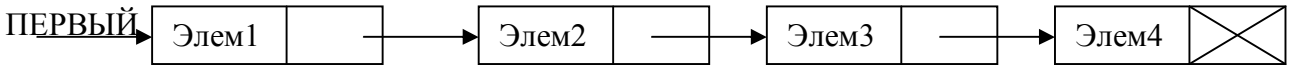
ПРЕДЫДУЩИЙ(A,x) - возвращает указатель на элемент, предшествующий элементу с указателем x или ПУСТО.

Списки.

Под списком будем понимать конечную последовательность элементов, взятых из некоторого множества.

Описание алгоритма часто включают некоторые списки, к которым добавляются и из которых удаляются элементы. Простейшим списком является строка, используемая для хранения последовательности символов $a_1, a_2, \ldots, a_n$.

Рассмотрим список Элем1, Элем2, Элем3, Элем4. Простейшей его реализацией будет структура последовательно связанных компонент.



Это можно реализовать в виде двух массивов, которые назовём ИМЯ и СЛЕДУЮЩАЯ. Если КОМПОНЕНТА – это индекс в рассматриваемом массиве, то ИМЯ[КОМПОНЕНТА] – сам элемент в этом массиве, а СЛЕДУЮЩАЯ[КОМПОНЕНТА] – индекс следующего элемента в списке (если такой существует).

	ИМЯ	СЛЕДУЮЩАЯ
0	–	1
1	Элем1	3
2	Элем4	0
3	Элем2	4

4	Элем3	2

Опишем процедуру, вставляющую новую компоненту в список. Предполагается, что СВОБОДНАЯ – номер незанятой ячейки в массивах ИМЯ и СЛЕДУЮЩАЯ, а ПОЗИЦИЯ – индекс той компоненты в списке, после которой надлежит вставить ЭЛЕМЕНТ.

ВСТАВИТЬ (ЭЛЕМЕНТ, СВОБОДНАЯ, ПОЗИЦИЯ)

ИМЯ[СВОБОДНАЯ] ← ЭЛЕМЕНТЫ

СЛЕДУЮЩАЯ[СВОБОДНАЯ] ← СЛЕДУЮЩАЯ[ПОЗИЦИЯ]

СЛЕДУЮЩАЯ[ПОЗИЦИЯ] ← СВОБОДНАЯ

Например: Вставить в рассматриваемый выше список один элемент, имеющий имя НовЭлем после Элем2 и получить список

Элем1, Элем2, НовЭлем, Элем3, Элем4

Решение:

ВСТАВИТЬ (НовЭлем, 5, 3)

ИМЯ[5] ← НовЭлем

СЛЕДУЮЩАЯ[5] ← СЛЕДУЮЩАЯ[3] {СЛЕДУЮЩАЯ[5]=4}

СЛЕДУЮЩАЯ[3] ← 5

	ИМЯ	СЛЕДУЮЩАЯ
0	–	1
1	Элем1	3
2	Элем4	0
3	Элем2	5
4	Элем3	2
5	НовЭлем	4

Стеки и очереди

Стеки и очереди использовались в математике и делопроизводстве в докомпьютерную эру. Кнут отмечает, что в 1947 году Тьюринг использовал стеки для связи подпрограмм.

Стеки и очереди – это динамические множества, в которых элемент, удаляемый из множества операцией УДАЛИТЬ, не задаётся произвольно, а определяется структурой множества. Именно, из стека можно удалить только элемент, который был добавлен в него последним, «последний пришёл – первый ушёл» (LIFO). Из очереди, напротив, можно удалять только те элементы, которые были внесены в множество первыми, «первый пришёл – первый ушёл» (FIFO). Существует несколько способов реализовать стеки и очереди.

Стеки

Стек или другое название «магазин» (аналогия с магазином автомата Калашникова) подобен стопке тарелок. Для реализации стека ёмкостью не более n элементов можно использовать массив A[1.. n]. Наряду с массивом необходимо хранить указатель на последний элемент стека ВЕРШИНА_СТЕКА[A]. Сам стек состоит из элементов A[1.. ВЕРШИНА_СТЕКА[A]], где A[1] – нижний элемент стека, а A[ВЕРШИНА_СТЕКА[A]] – верхний элемент, или вершина стека.

Если ВЕРШИНА_СТЕКА[A] = 0, то стек пуст. Если ВЕРШИНА_СТЕКА[A] = n, то добавление элементов в стек не возможно, т.к. стек полон.

Например, стек, состоящий из трёх элементов Элем1, Элем2, Элем3 можно представить в виде массива A:

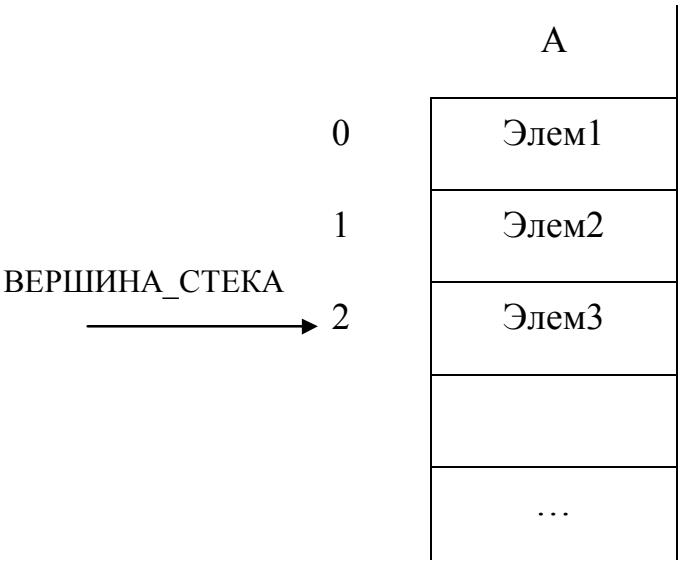


Рис.4 Реализация стека

Операции со стеком (проверка пустоты, добавление элемента, удаление элемента) записываются так:

ПУСТ_ЛИ_СТЕК(A)
если ВЕРШИНА_СТЕКА[A] = 0
тогда вернуть ИСТИНА
иначе вернуть ЛОЖЬ

ДОБАВИТЬ_В_СТЕК(A, x)
ВЕРШИНА_СТЕКА[A] ← ВЕРШИНА_СТЕКА[A] + 1
A[ВЕРШИНА_СТЕКА[A]] ← x

УДАЛИТЬ_ИЗ_СТЕКА(A)
если ПУСТ_ЛИ_СТЕК(A)
тогда вывести «Стек пуст»
иначе ВЕРШИНА_СТЕКА[A]←ВЕРШИНА_СТЕКА[A] - 1
вернуть A[ВЕРШИНА_СТЕКА[A] + 1]

Каждая из трёх операций со стеком выполняется за время O(1).

Очереди

У очереди есть *голова* и есть *хвост*. Элемент, добавляемый в очередь, оказывается в её хвосте, элемент, удаляемый из очереди, находится в её голове.

Реализовать очередь можно на базе массива A[1..n]. при этом необходимо хранить указатели на голову и хвост очереди (ГОЛОВА[A], ХВОСТ[A]). Очередь пуста, если ГОЛОВА[A] = ХВОСТ[A] = 1. Очередь заполнена, если ГОЛОВА[A] = ХВОСТ[A] + 1 (предполагается что массив свёрнут в кольцо).

1	2	3	4	5	6	7	8	9	10	11	12
						15	6	9	8	4	
						↑				↑	

ГОЛОВА[A]=7 ХВОСТ[A]=12

1	2	3	4	5	6	7	8	9	10	11	12
3	5					15	6	9	8	4	17
	↑					↑					

ХВОСТ[A]=3 ГОЛОВА[A]=7

ДОБАВИТЬ_В_ОЧЕРЕДЬ(A, x)

A[ХВОСТ[A]] ← x

если ХВОСТ[А] = n

тогда ХВОСТ[А] ← 1

иначе ХВОСТ[А] ←ХВОСТ[А] + 1

УДАЛИТЬ_ИЗ_ОЧЕРЕДИ(А)

x ← А[ГОЛОВА[А]]

если ГОЛОВА[А] = n

тогда ГОЛОВА[А] ← 1

иначе ГОЛОВА[А] ←ГОЛОВА[А] + 1

вернуть x

Каждая из этих процедур работает за время O(1).

Связанные списки

Структуры данных, основанные на указателях относятся к «фольклору». Согласно Кнуту, указатели использовались ещё в первых компьютерах с магнитными барабанами.

В связанном списке элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Списки являются удобным способом хранения динамических множеств, позволяющих реализовать все операции.

Двусторонний связанный список – это запись, содержащая три поля: КЛЮЧ, СЛЕДУЮЩИЙ и ПРЕДЫДУЩИЙ. СЛЕДУЮЩИЙ(x) = ПУСТО, если x – последний элемент списка, ПРЕДЫДУШИЙ(x) = ПУСТО, если x – первый элемент списка. Мы предполагаем, что известен указатель на голову списка – ГОЛОВА[А]. Если ГОЛОВА[А] = ПУСТО, то список пуст.

Односторонние связные списки – это записи, в которых отсутствует поле ПРЕДЫДУЩИЙ. В **упорядоченном** списке элементы расположены в порядке возрастания ключей, в противном случае список **неупорядочен**. В **кольцевом списке** поле ПРЕДЫДУЩИЙ головы списка указывает на хвост, а поле СЛЕДУЮЩИЙ хвоста списка указывает на голову.

Работа со списками.

Рассмотрим основные процедуры обработки списка. К ним относятся процедуры поиска, добавления и удаления элемента.

НАЙТИ_В_СПИСКЕ(А, к)
x ← ГОЛОВА[A]
пока x ≠ ПУСТО и КЛЮЧ[x] ≠ к
    делать x ← СЛЕДУЮЩИЙ[x]
вернуть x

Поиск в списке из n элементов требует в худшем случае (когда приходится просматривать весь список) O(n) операций.

ДОБАВИТЬ_В_СПИСОК(А, x)
СЛЕДУЮЩИЙ[x] ← ГОЛОВА[A]
если ГОЛОВА[A] ≠ ПУСТО
    тогда СЛЕДУЮЩИЙ[ГОЛОВА[A]] ← x
ГОЛОВА[A] ← x
СЛЕДУЮЩИЙ[x] ← ПУСТО

Эта процедура выполняется за время O(1) (не зависящее от длины списка).

УДАЛИТЬ_ИЗ_СПИСКА(А, x)

если ПРЕДЫДУЩИЙ[x] ≠ ПУСТО

тогда СЛЕДУЮЩИЙ[ПРЕДЫДУЩИЙ[x]] ← СЛЕДУЮЩИЙ[x]

иначе ГОЛОВА[A] ← СЛЕДУЮЩИЙ[x]

если СЛЕДУЮЩИЙ[x] ≠ ПУСТО

тогда ПРЕДЫДУЩИЙ [СЛЕДУЮЩИЙ [x]] ← ПРЕДЫДУЩИЙ[x]

Удаление элемента из списка происходит за время O(1), однако при удалении элемента с заданным ключом, его необходимо сначала найти, что потребует время O(n).

Если забыть об особых ситуациях на концах списка, процедуру удаления можно записать совсем просто:

УДАЛИТЬ_ИЗ_СПИСКА (A, x)

СЛЕДУЮЩИЙ[ПРЕДЫДУЩИЙ[x]] ← СЛЕДУЮЩИЙ[x]

ПРЕДЫДУЩИЙ [СЛЕДУЮЩИЙ [x]] ← ПРЕДЫДУЩИЙ[x]

Такие упрощения станут законными, если добавить к списку A фиктивный элемент ПУСТО[A], который будет иметь поля СЛЕДУЮЩИЙ и ПРЕДЫДУЩИЙ как и в других элементах списка. Этот элемент называемый «часовым» не позволит нам выйти за пределы списка. Указатель на него играет роль значения ПУСТО. Замкнём список в кольцо. При этом СЛЕДУЮЩИЙ[ПУСТО[A]] – указатель на голову списка, так что атрибут ГОЛОВА[A] становится лишним. Тогда процедура поиска будет выглядеть так:

НАЙТИ_В_СПИСКЕ'(A, к)

x ← СЛЕДУЮЩИЙ[ПУСТО[A]]

пока x ≠ ПУСТО[A] и КЛЮЧ[x] ≠ к

делать x ← СЛЕДУЮЩИЙ[x]

вернуть x

А процедура удаления будет выглядеть так:

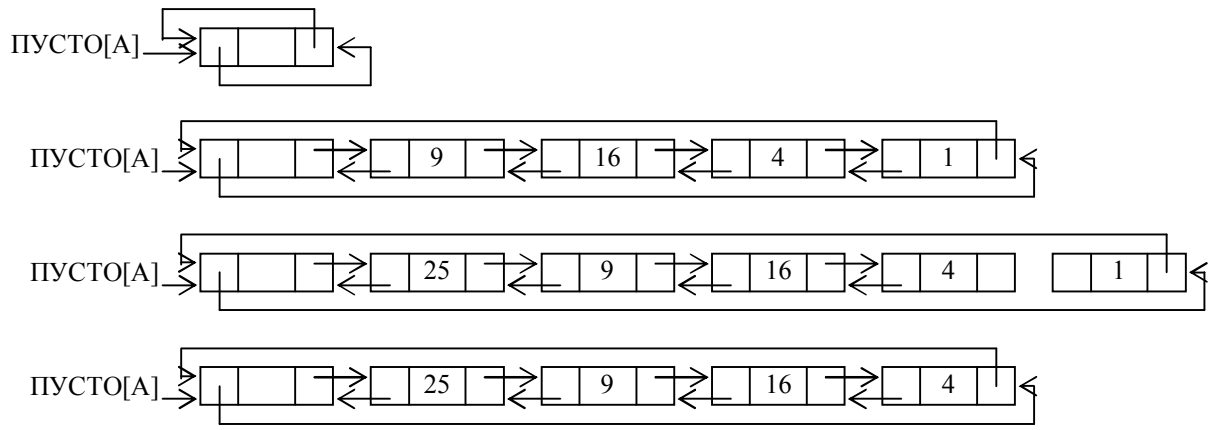
УДАЛИТЬ_ИЗ_СПИСКА(A, x)

СЛЕДУЮЩИЙ[x] ← СЛЕДУЮЩИЙ[ПУСТО[A]]

ПРЕДЫДУЩИЙ [СЛЕДУЮЩИЙ [ПУСТО[A]]] ← ПРЕДЫДУЩИЙ[x]

СЛЕДУЮЩИЙ [ПУСТО[A]] ← x

ПРЕДЫДУЩИЙ[x] ← ПУСТО[A]



Использование фиктивных элементов едва ли улучшает время работы алгоритма, но упрощает программу. Не следует применять фиктивные элементы без нужды. Если алгоритм использует много коротких списков, использование фиктивных элементов может увеличить объём используемой памяти.

- Реализация указателей и записей с несколькими полями.

В некоторых языках программирования не предусмотрено ни указателей, ни элементов имеющих несколько полей. В таком случае приходится обходиться массивами, используя индекс в массиве как указатель и заменяя массив записей несколькими массивами.

- Представление с помощью нескольких массивов.

Массив, составленный из записей, можно записать несколькими массивами (по одному на каждое поле записей). Например, предыдущий список можно представить с помощью трёх массивов:

СЛЕДУЮЩИЙ		3	/		2		5	
КЛЮЧ		4	1		16		9	
ПРЕДЫДУЩИЙ		5	2		7		/	

А 7

Каждому элементу списка соответствует тройка (КЛЮЧ[x], СЛЕДУЮЩИЙ[x], ПРЕДЫДУЩИЙ[x]) для некоторого индекса x. в качестве ПУСТО можно использовать число не являющееся индексом никакого элемента массивов.

В наших программах обозначение типа КЛЮЧ[x] может трактоваться двояко: и как элемент массива КЛЮЧ с индексом x, и как поле КЛЮЧ записи с адресом x (в зависимости от возможностей языка программирования полезна та или другая трактовка).

- Представление с помощью одного массива.

Вместо нескольких массивов можно использовать один, размещая в нём различные поля одного объекта рядом. Так обычно поступает

компилятор: если в программе используется массив элементов, каждый из которых имеет несколько полей, то на каждый элемент отводится непрерывный участок памяти, в котором друг за другом размещаются значения полей. Указателем на элемент обычно считают адрес первой ячейки этого участка; адреса полей получают сдвигом на определённые константы.

			4	7	13	1	/	4				16	4	19				9	13	/			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

A 19

Такое представление позволяет хранить в одном массиве записи разных типов, отводя под них участки разной длины.

Мультисписки

Списки могут быть не только линейными, но и кольцевыми. В кольцевом списке для последнего элемента следующим является первый, а если список двунаправленный, то для первого предыдущим является последний. Как и линейный, кольцевой список определяется указателем на свой первый элемент. При просмотре линейного списка переход от одного элемента к другому осуществляется до тех пор, пока рабочий указатель не станет равным Nil. При просмотре кольцевого списка надо переходить к следующему элементу до тех пор, пока рабочий указатель не совпадёт с указателем на первый элемент. Поэтому первый элемент должен обрабатываться отдельно, а просмотр при помощи цикла следует начинать со второго элемента.

Мультисписки представляют собой структуру, каждый элемент которой входит в более чем один список одновременно и имеет

соответствующее числу списков количество полей связи. Часто в виде мультисписков представляют матрицы очень большой размерности, в которых большинство элементов равны 0 (такие матрицы называются разреженными). Мультисписки обеспечивают эффективное хранение таких структур в памяти: хранятся только те элементы, которые отличны от нуля. Каждый элемент входит в два списка: список-строку, список-столбец. Вся матрица представлена m + n списками, где m и n – соответственно число строк и число столбцов матрицы, каждый элемент хранит значение элемента матрицы и две ссылки: на следующий элемент в строке и на следующий элемент в столбце, указатели на начала этих списков хранятся в двух массивах. Описание типа данных одного элемента матрицы –мультисписка аналогично описанию элемента-очереди или узла дерева. Для описания матрицы потребуется два массива – массив указателей на первые элементы списков-строк и на первые элементы списков-столбцов.

Управление памятью

В работе компьютерных систем нередко возникают ситуации, когда ограниченными ресурсами основной памяти приходится управлять, т.е. разделять между несколькими совместно использующими её «конкурентами». Различные языки программирования по-своему реализуют данную задачу, что связано с размерами объектов и выбранной стратегией. Ещё один важный вопрос, возникающий при управлении памятью – фрагментация памяти. Алгоритмы, решающие эти проблемы называются «сборкой мусора».

Выделение и освобождение памяти

При добавлении нового элемента в список надо отвести под него место в памяти. Стало быть, необходимо вести учёта использования адресов. В некоторых системах этим ведает специальная подпрограмма – **сборщик мусора**, которая определяет, какие участки памяти более не используются и возвращает их для повторного использования. Во многих случаях можно возложить обязанности по выделению и освобождению памяти на саму структуру данных.

Пусть массивы, с помощью которых мы представляем наш двусторонне связанный список, имеют длину m, и пусть в данный момент в списке содержится $n \leq m$ элементов. Остальные n-m мест в массиве свободны.

Мы будем хранить свободные позиции в односторонне связанном списке, называемом **списком свободных позиций.** Этот список использует только массив СЛЕДУЮЩИЙ. Голова списка хранится в переменной СВОБОДНЫЙ. Список свободных позиций хранится вперемежку со списком А.

Список свободных позиций ведёт себя как стек: из всех свободных участков памяти под новую запись выделяется тот, который был освобождён последним. Поэтому для выделения и освобождения памяти можно воспользоваться реализацией стековых операций ДОБАВИТЬ_В_СТЕК и УДАЛИТЬ_ИЗ_СТЕКА на базе списка. Рассмотрим процедуры ВЫДЕЛИТЬ_МЕСТО и ОСВОБОДИТЬ_МЕСТО, в переменной СВОБОДНЫЙ хранится индекс первой в списке свободной позиции.

ВЫДЕЛИТЬ_МЕСТО()
если СВОБОДНЫЙ = ПУСТО
тогда ошибка “Свободного места нет”
иначе x ← СВОБОДНЫЙ
СВОБОДНЫЙ ← СЛЕДУЮЩИЙ[x]
вернуть x

ОСВОБОДИТЬ_МЕСТО(x)
СЛЕДУЮЩИЙ[x] ← СВОБОДНЫЙ
СВОБОДНЫЙ ← x

Пример:

СЛЕДУЮЩИЙ КЛЮЧ	/	3	/	8	2	1	5	6
ПРЕДЫДУЩИЙ		4	1		16		5	
		5	2		7		/	

СВОБОДНЫЙ	4
A	7

СЛЕДУЮЩИЙ	/	3	/	7	2	1	5	6
КЛЮЧ								
ПРЕДЫДУЩИЙ		4	1	25	16		5	
		5	2	/	7		4	

СВОБОДНЫЙ	8
А	4

СЛЕДУЮЩИЙ	/	3	/	7	8	1	2	6
КЛЮЧ								
ПРЕДЫДУЩИЙ		4	1	25			5	
		7	2	/			4	

СВОБОДНЫЙ	5
А	4

Часто один список свободных позиций обслуживает сразу несколько динамических множеств.

СЛЕДУЮЩИЙ	5	/	6	8	/	2	1	/	7	4
КЛЮЧ										
ПРЕДЫДУЩИЙ	К1	К2	К3		К5	К6	К7		К9	
	7	6	/		1	3	9		/	

СВОБОДНЫЙ	10
A1	9
A2	3

Описанные процедуры выделения и освобождения памяти просты и удобны (работают за время O(1)). Их можно приспособить для хранения однотипных записей любого вида, отведя одно из полей записи под хранение индекса СЛЕДУЮЩИЙ (в свободных позициях).

Система управления памятью со сборкой мусора.

Блоки фиксированной длины.

При возврате в свободную память целых сцепленных структур должно освобождаться большое количество звеньев, среди которых могут находиться ещё нужные, скажем входящие в другую, ещё нужную структуру.

Поэтому, идея сборки мусора такова: сначала отмечаются всё ещё нужные звенья, а затем из того, что осталось формируется новый список.

Для отметки нужно пройти по всем указателям, отмечая по пути все звенья, пока не попадём либо уже в отмеченное, либо с пустым указателем.

Все указатели на ещё нужные сцепленные структуры собраны вместе в вектор или список (Нужные(1, КН)). С этих указателей и будет начинаться процесс разметки.

Для того чтобы различать нужные звенья и ненужные звенья придется отвести дополнительный бит для признака и две вспомогательные программы для работы с ним.

Указатель:

Признак	Следующий

Указатель необходим лишь для организации списка свободной памяти и может использоваться программой пользователя произвольным способом, т.е. фактически не происходит потери памяти по организации её в структуру линейного списка.

Поле признака необходимо при сборке мусора - ненужной информации - для восстановления свободной памяти и не должно использоваться программой пользователя. Более предпочтительным решением было бы поставить в соответствие каждому звену (блоку) памяти 1 бит и все эти биты собрать вместе в отдельной области памяти.

Программы для реализации системы управления памятью со сборкой мусора:

1. СЛЕДУЮЩИЙ (x)
2. УСТАНОВИТЬ_СЛЕДУЮЩИЙ(x, a)
3. ПУСТО(x) - логическая функция, которая в результате выдаёт "истинно", если звено отмечено как нужное, "ложь" - если нет.
4. УСТАНОВИТЬ_ПРИЗНАК(x, к) - устанавливает признак в звене x равный к.
5. ПОДГОТОВИТЬ - заполняет массив НУЖНЫЕ пустыми указателями и устанавливает в указатель на свободную память пустое значение, т.е. программа готовит память для последующей её организации в список свободной памяти, которая произойдёт при первом запросе памяти.

ПОДГОТОВИТЬ

1. FR ← ПУСТО
2. **для** p **от** 1 **до** КН **с шагом** 1
3. **делать**
4. НУЖНЫЕ(p) ← ПУСТО

5. **конец делать**
6. **конец**

6. ВЫДЕЛИТЬ

ВЫДЕЛИТЬ

1. **если** FR = ПУСТО **тогда** СОБРАТЬ_МУСОР
2. result ← FR
3. FR ← СЛЕДУЮЩИЙ (FR)
4. **конец**

5. СОБРАТЬ_МУСОР - размечает все нужные звенья, а из оставшихся формирует новый список свободной памяти.

СОБРАТЬ_МУСОР

1. **для** p **от** 1 **до** КН **с шагом** 1
2. **делать** РАЗМЕТИТЬ (НУЖНОЕ (p))
7. **для** p **от** 1 **до** N с **шагом** 1
3. **делать**
4. **если** НУЖНОЕ(p) **тогда** УСТАНОВИТЬ_ПРИЗНАК (p, false)
5. **иначе делать**
6. УСТАНОВИТЬ_СЛЕДУЮЩИЙ(p, FR)
8. FR ← p
9. **Конец делать**
10. **если** FR ← ПУСТО **тогда** *ошибка*
11. **конец**

6. РАЗМЕТИТЬ(p), где p - указатель на отдельное звено в сцепленной структуре. Процедура отмечает все звенья в структуре, на которые указывает этот указатель p.

```
РАЗМЕТИТЬ(p)
    если p = ПУСТО или НУЖНЫЕ(p), тогда выход
                                   иначе
        делать
                УСТАНОВИТЬ_ПРИЗНАК (p, true)
                Пройти по указателям, имеющимся в данном
                звене, т.е. рекурсивно использовать эту же
                программу к указателям в звене
        Конец делать
    Конец
```

Недостатком этого метода является то, что программа РАЗМЕТИТЬ рекурсивна, следовательно, должна использоваться дополнительная память для запоминания своих промежуточных результатов. А так как нет возможности предсказать заранее, сколько потребуется памяти для этого и нельзя брать память из области свободной памяти, то могут возникнуть трудности с обеспечением работы этой программы. Следовательно, можно разработать другую программу, которая обходилась бы без дополнительной памяти, а использовала часть проверяемых звеньев для запоминания текущей информации. Такие программы работают значительно медленнее, чем простые рекурсивные программы и к ним прибегают в крайнем случае.

Другой способ обеспечения возврата памяти позволяет обойтись без фактического использования списка свободной памяти. После того, как все ненужные звенья стали отмеченными, не строят списка свободной памяти, а запоминают указатель на первое нужное звено.

Если требуется новое звено, то производится поиск, начиная с того места, где находится (куда ведёт) этот указатель до обнаружения следующего звена, которое отмечено как ненужное (свободное). Оно и выдаётся

пользователю, а указатель на свободное звено сдвигается таким образом, чтобы указывать на следующий за выбранным. Когда исчерпывается вся свободная память, выполняется новый процесс разметки нужных звеньев.

Этот метод, однако, не короче и не эффективнее предыдущего, он иначе распределяет затрачиваемое время. Время, которое уходило на формирование списка свободной памяти затрачивается теперь постепенно, по мере возникновения надобности в новых звеньях.

Система управления памятью для блоков различной длины.

Рассматриваемый способ управления памятью также включает в себя сборку мусора, когда имеющаяся свободная память исчерпана. Идея сборки мусора иная:

Нужную память сначала размечают, затем сдвигают к началу, чтобы закрыть все промежутки, представляющие собой мусор. "Двигая" звенья по памяти придется менять их адреса, следовательно, меняются все указатели.

Всю работу по сборке мусора разбивают на этапы:

1. Пометить все звенья, к которым отрыт доступ через внешние указатели.
2. Перебирая все звенья памяти определить для каждого нужного звена место, куда его надо будет сдвинуть для ликвидации промежутков. Для хранения этой информации используется поле СЛЕДУЮЩИЙ
3. Перебирая подряд все звенья памяти в каждом звене каждый указатель заменить на значение поля СЛЕДУЮЩИЙ того звена, на которое он указывает, изменить и все внешние указатели.
4. Сдвинуть все помеченные нужные звенья в начало области памяти, ликвидируя промежутки и уничтожая разметку звеньев. Указатель свободной памяти установить непосредственно за последним сдвинутым

звеном. Так как звенья могут иметь различную длину и по разному интерпретироваться, будем считать, что в каждом звене есть два дополнительных поля:

- для указания длины звена
- для типа звена

С каждым полем связано по две программы:

1) СЛЕДУЮЩИЙ (x)
2) УСТАНОВИТЬ_СЛЕДУЮЩИЙ (x, p)
3) НУЖНЫЙ (x)
4) УСТАНОВИТЬ_ПРИЗНАК (x, L), где L принимает значения True или False.
5) ДЛИНА (x)
6) УСТАНОВИТЬ_ДЛИНУ (x, к) - устанавливает значение поля длины данного звена x равным к.
7) ТИП_ЗВЕНА (x) - результатом работы функции является значение типа данного звена
8) УСТАНОВИТЬ_ТИП (x, T) - устанавливает значение поля типа данного звена x равным T.
9) ПОДГОТОВИТЬ - готовит область памяти для дальнейшей обработки.

ПОДГОТОВИТЬ

ПУСТО ← 0

FR ← 1

для p **от** 1 **до** КН **с шагом** 1

делать

НУЖНЫЕ (p) ← ПУСТО

Конец делать

КД ← количество служебных полей в каждом блоке

конец

10) ВЫДЕЛИТЬ (к, Т) - для выделения области оперативной памяти, к - длина области оперативной памяти, область типа Т.

ВЫДЕЛИТЬ (к, Т)

если FR + к + КД > N **тогда** СОБРАТЬ_МУСОР

Result ← FR

УСТАНОВИТЬ_ДЛИНУ (FR, к)

УСТАНОВИТЬ ТИП (FR, Т)

FR ← FR + к + КД

конец

11) РАЗМЕТИТЬ (р) - размечает сцепленную структуру с указателем р.

РАЗМЕТИТЬ (р)

если р ≠ ПУСТО или НУЖНЫЕ (р) **тогда**

УСТАНОВИТЬ_ПРИЗНАК (р, true)

Конец

Пройти по указателям, имеющимся в данном звене (используя поле типа), т.е. рекурсивно используя подпрограмму разметить, отметить все те звенья, в которые ведут эти указатели.

12) СОБРАТЬ_МУСОР - размечает все нужные звенья, планирует перемещения, корректирует указатели, перемещает нужные звенья в начало области памяти и корректирует указатели свободной памяти.

СОБРАТЬ_МУСОР

для р **от** 1 **до** КН **с шагом** 1

делать

РАЗМЕТИТЬ (НУЖНЫЕ(p))

конец делать

p ← 1

Q ← 1

пока p < FR **делать**

если НУЖНЫЕ (p) **тогда**

делать

УСТАНОВИТЬ_СЛЕДУЮЩИЙ (p, Q)

Q ← Q + ДЛИНА (p) + КД

конец делать

p ← p + ДЛИНА (p) + КД

конец делать

если Q + КД + к > N **тогда**

переполнение, память не может быть выделена

иначе

делать

для p **от** 1 **до** КН **с шагом** 1

делать

НУЖНЫЕ (p)←СЛЕДУЮЩИЙ (НУЖНЫЕ (p))

конец делать

p ← 1

пока p < FR **делать**

если НУЖНЫЕ (p) **тогда**

выделить все поля указателей в звене p и скорректировать (обновить) их, используя поле СЛЕДУЮЩИЙ того звена, в

которое он указывает и так рекурсивно следовать по всем указателям;

p ← p + ДЛИНА (p) + КД

конец делать

p ← 1

пока p < FR **делать**

если НУЖНОЕ (p) **тогда**

делать

УСТАНОВИТЬ_ПРИЗНАК (p, false)

для i **от** 1 **до** ДЛИНА (p)+КД-1 **с шагом** 1

делать

MEMORY(СЛЕДУЮЩИЙ(p)

+ i ← MEMORY (p + i)

конец делать

конец делать

p ← p + ДЛИНА(p) КД

конец делать

FR ← Q

конец делать

конец

Объектно-ориентированное программирование

В основе того или иного языка программирования лежит некоторая руководящая идея, оказывающая существенное влияние на стиль соответствующих программ. Исторически первой была идея процедурного структурирования, в соответствии с которой программист должен бал решить какие именно процедуры он будет использовать в своей программе, а затем выбрать наилучшие алгоритмы для их реализации. Типичным представителем является Фортран. По мере прогресса в области вычислительной математики, акцент в программировании стал смещаться с процедур в сторону организации данных. Паскаль, Си, Ада имеют более или менее развитые структуры типов данных. Логическим следствием развития этого направления стал модульный подход к разработке программ, характеризующийся стремлением «спрятать» данные и процедуры внутрь модуля.

Начиная с языка Симула 67, в программировании наметился новый подход, который получил название объектно-ориентированного программирования (ООП). Его руководящей идеей является стремление связать данные с обрабатывающими эти данные процедурами в единое целое – объект. Фактически ООП можно рассматривать как модульное программирование нового уровня, когда вместо, во многом случайного, механического объединения в модуле процедур и данных акцент делается на смысловую связь полей и методов объекта.

Объектно-ориентированные языки программирования пользуются в последнее время большой популярностью среди программистов, так как они позволяют использовать преимущества объектно-ориентированного подхода

не только на этапах проектирования и конструирования программных систем, но и на этапах их реализации, тестирования и сопровождения.

Обзор объектно-ориентированных языков программирования.

Первый объектно-ориентированный язык программирования Simula 67 был разработан в конце 60-х годов в Норвегии. Авторы этого языка очень точно угадали перспективы развития программирования: их язык намного опередил свое время. Однако современники (программисты 60-х годов) оказались не готовы воспринять ценности языка Simula 67, и он не выдержал конкуренции с другими языками программирования (прежде всего, с языком Fortran). Прохладному отношению к языку Simula 67 способствовало и то обстоятельство, что он был реализован как интерпретируемый (а не компилируемый) язык, что было совершенно неприемлемым в 60-е годы, так как интерпретация связана со снижением эффективности (скорости выполнения) программ.

Но достоинства языка Simula 67 были замечены некоторыми программистами, и в 70-е годы было разработано большое число экспериментальных объектно-ориентированных языков программирования: например, языки CLU, Alphard, Concurrent Pascal и др. Эти языки так и остались экспериментальными, но в результате их исследования были разработаны современные объектно-ориентированные языки программирования: C++, Smalltalk, Eiffel и др.

Наиболее распространенным объектно-ориентированным языком программирования безусловно является C++. Свободно распространяемые коммерческие системы программирования C++ существуют практически на любой платформе. Широко известна свободно распространяемая система программирования G++, которая дает возможность всем желающим

разобрать достаточно хорошо и подробно прокомментированный исходный текст одного из образцовых компиляторов языка C++. Завершается работа по стандартизации языка C++: последний Draft стандарта C++ выпущен в июне 1995 г. (он доступен по Internet).

Разработка новых объектно-ориентированных языков программирования продолжается. С 1995 года стал широко распространяться новый объектно-ориентированный язык программирования Java, ориентированный на сети компьютеров и, прежде всего, на Internet. Синтаксис этого языка напоминает синтаксис языка C++, однако эти языки имеют мало общего. Java интерпретируемый язык: для него определены внутреннее представление (bytecode) и интерпретатор этого представления, которые уже сейчас реализованы на большинстве платформ. Интерпретатор упрощает отладку программ, написанных на языке Java, обеспечивает их переносимость на новые платформы и адаптируемость к новым окружениям. Он позволяет исключить влияние программ, написанных на языке Java, на другие программы и файлы, имеющиеся на новой платформе, и тем самым обеспечить безопасность при выполнении этих программ. Эти свойства языка Java позволяют использовать его как основной язык программирования для программ, распространяемых по сетям (в частности, по сети Internet).

Объектно-ориентированный подход.

До сих пор, составляя сложные программы, мы оставались в рамках *процедурного* подхода. Основная задача разбивалась на ряд более простых задач, каждая из которых могла быть оформлена в виде отдельной процедуры или функций. При этом подходе больше внимания уделяется разработке алгоритма и существенно меньше — используемым структурам данных.

В настоящее время для разработки сложных программ все больше применяется *объектно-ориентированный* подход.

Основополагающей идеей объектно-ориентированного программирования является объединение данных и обрабатывающих их программ в единое целое – объекты. Объектно-ориентированное программирование – это методология программирования, которая основана на представлении программы в виде совокупности объектов, каждый из которых является реализацией определённого класса (типа особого виды), а классы образуют иерархию на принципах наследуемости.

Концепция объектно-ориентированного программирования подразумевает, что основой управления процессом реализации программы является передача сообщений объектам. Объект имеет состояние (например, размер, цвет и т.д.) и методы (действия, на которые способен объект). При этом объект характеризуется как совокупность допустимых для данного объекта действий. Чтобы заставить объект что-то сделать, нужно послать ему соответствующее сообщение.

Состояние объекта характеризуется значениями полей; методы объекта — связанные с ним процедуры и функции, которым доступны поля. Передача сообщений экземпляру объекта происходит в виде вызовов его методов с заданными параметрами.

Таким образом, объектно-ориентированная программа – это данные, управляющие доступом к коду, а объектно-ориентированное программирование – способом разработки программ, которые состоят из объектов, отдельных фрагментов кода, обрабатывающего данные, взаимодействующие друг с другом через определённые интерфейсы.

Основные определения компонент ООП:

Класс – тип объектов, который при помощи описания класса, определяющего переменные состояния и протокол доступа к объектам данного класса.

Объект – инкапсулированная абстракция, включающая информацию о состоянии и чётко определённое множество протокола доступа (сообщения, которые обрабатывает объект).

Сообщение – специальный символ, идентификатор или ключевое слово с (или без) параметрами, которое представляет выполняемое объектом действие.

Метод – ряд выражений, которые определяют реакцию объекта на сообщение, определённое для некоторого класса. Объявление методов включено в описание объекта. Возможность управлять состояниями объекта посредством вызова в итоге и определяет поведение объекта. Эту совокупность методов часто называют интерфейсом объекта.

Объектно-ориентированные языки программирования характеризуются тремя основными свойствами:

Инкапсуляция (encapsulation) – объединение в одном объекте данных и методов их обработки. При этом данные одного объекта недоступны методам другого объекта, если эти объекты не связаны друг с другом. При использовании инкапсуляции данные, принадлежащие объекту, защищаются от возможных ошибок, которые могут возникнуть при прямом доступе к этим данным. Кроме того, применение этого принципа очень часто помогает локализовать возможные ошибки в коде программы. а это намного упрощает процесс поиска и исправления этих ошибок. Можно сказать, что

инкапсуляция подразумевает под собой скрытие данных, что позволяет защитить эти данные. Однако применение инкапсуляции ведёт к снижению эффективности доступа к элементам объекта. Это обусловлено необходимостью вызова методов для изменения внутренних элементов (переменных) объекта. Но при современном уровне развития вычислительной техники эти потери в эффективности не играют существенной роли.

Предположим, мы хотим дополнить уже разработанный объектный тип новыми свойствами. Можно, конечно, его переделать, но объектно-ориентированный подход рекомендует создать новый объектный тип (объект-наследник), который является расширением уже имеющегося *базового* типа. Новый тип будет включать в себя все данные и методы базового типа. И при этом может иметь собственные поля, собственные методы и перекрывать своими методами одноименные унаследованные методы. Таким образом, в ООП работает механизм *наследования* — такого "отношения между объектами, когда один объект повторяет структуру и поведение другого" (Гради Буч). Базовые объекты называют *родителями* и *предками,* а объекты-наследники — *потомками.* В результате реализации механизма наследования между объектами возникает *иерархия.*

Наследование (inheritance) – создание новых объектов на базе ранее определённых. Новые объекты-потомки сохраняют свойства своих родителей и обладают специфическими свойствами. Наследник называется порождённым (дочерним) типом, а тип, наследующий свои характеристики называется порождающим (родительским) типом. Тип «наследник» иногда называется производным типом.

Правила наследования:

1. Информационные поля и методы родительского типа наследуются всеми его типами-потомками независимо от числа промежуточных уровней иерархии;

2. Доступ к полям и методам родительских типов в рамках описания любых типов потомков выполняется так, как будто бы они описаны в самом типе-потомке;

3. Ни в одном из типов потомков не могут использоваться идентификаторы полей, совпадающие с идентификаторами полей какого-либо из родительских типов. Это правило относится и к идентификаторам формальных параметров, указанных в заголовке методов;

4. Тип-потомок может доопределить произвольное число собственных методов и информационных полей;

5. Любое изменение текста в родительском методе автоматически оказывает влияние на все методы порождённых типов-потомков, которые его вызывают;

6. В противоположность информационным полям идентификаторы методов в типах-потомках могу совпадать с именами методов в родительских типах. При этом одноимённый метод в типе-потомке подавляет одноимённый ему родительский – и в рамках типа-потомка при указании имени такого метода будет вызываться именно метод типа-потомка, а не родительский.

Полиморфизм (polymorphism) – возможность замещения методов объекта-родителя одноименными методами объекта-потомка. В общем смысле концепцией полиморфизма является идея «один интерфейс, множество методов». Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного интерфейса для единого класса действий. Выбор конкретного действия в зависимости от ситуации возлагается на компилятор.

Механизм работы ООП в таких случаях можно описать примерно так: при вызове того или иного метода класса сначала ищется метод у самого класса. Если метод найден, то он выполняется, и поиск этого метода на этом завершается. Если метод не найден, то обращаемся к родительскому классу и ищем вызванный метод у него. Если он найден, поступаем как при нахождении метода в самом классе. А если нет, продолжаем дальнейший поиск вверх по иерархическому дереву, вплоть до корня (верхнего класса) иерархии. Этот пример отражает так называемый «механизм раннего связывания».

Большинство современных программ предполагают интерактивный режим работы. В этом случае все действия программы являются ответом на те или иные действия пользователя — программа реагирует на внешнее событие: нажатие клавиши на клавиатуре, кнопки на мышке, перемещение мышки и т.п. Обработкой всех этих событий при традиционном программировании обычно занимается главная программа. Объектно-ориентированный подход предполагает создание в каждом из объектных типов специальной процедуры HandleEvent — *обработчика событий.* Тем самым "убиваются два зайца": во-первых, сохраняется принцип инкапсуляции и, во-вторых, упрощается доступ к элементам данных и методам объекта. Что же остается в этом случае на долю главной

программы? Так называемый цикл опроса клавиатуры и вызов обработчика событий для конкретного объекта.

Примеры алгоритмов, использующих объектно-ориентированный подход:

Пример 1:

Рассмотрим задачу линейного массива на основе объектно-ориентированного подхода. Методы: ввод, вывод, сортировка.

ОПИСАНИЯ

Тип МАС – массив [1..n] заданного типа

МАССИВ – **объект**, состоящий из:

Переменных состояния: А – типа МАС и n – размерность массива

Методов: ВВОД, ВЫВОД, СОРТИРОВКА

{описание методов}

МАССИВ.ВВОД;

Ввести размерность массива

В цикле ввести значения массива

МАССИВ.ВЫВОД;

В цикле вывести значения массива

МАССИВ.СОРТИРОВКА;

Организация обменной сортировки массива.

ОСНОВНАЯ ПРОГРАММА

Пусть а – это массив типа МАССИВ, тогда

а.ВВОД;

a.ВЫВОД;

a.СОРТИРОВКА;

a.ВЫВОД

КОНЕЦ

Пример 2:

Реализовать класс для обработки двумерных массивов, используя ранее разработанный класс для работы с линейными массивами. Методы: ВВОД, ВЫВОД.

ОПИСАНИЯ

Тип МАС – массив [1..n] заданного типа

МАССИВ – **объект**, состоящий из:

Переменных состояния: A – типа МАС и n – размерность массива

Методов: КОЛ_ЭЛЕМ, ВВОД, ВЫВОД

{создаём объектный тип для двумерного массива на основе объекта – линейного массива}

МАССИВ2 – **объект**, наследник объекта МАССИВ, состоящий из:

b - массив типа МАС;

Методов: ВВОД, ВЫВОД

{описание методов}

МАССИВ. КОЛ_ЭЛЕМ;

Ввести размерность массива

МАССИВ.ВВОД;

В цикле ввести значения массива

МАССИВ.ВЫВОД;

В цикле вывести значения массива

МАССИВ2.ВВОД;

КОЛ_ЭЛЕМ;

для i=1 **до** n **делать**

МАССИВ.ВВОД;

b[i] ← a

конец делать

МАССИВ2.ВЫВОД;

для i=1 **до** n **делать**

a ← b[i]

МАССИВ.ВЫВОД;

конец делать

ОСНОВНАЯ ПРОГРАММА

Пусть a – это массив типа МАССИВ2, тогда

a.ВВОД;

a.ВЫВОД;

МАССИВ2.ВВОД;

КОЛ_ЭЛЕМ;

для i=1 **до** n **делать**

МАССИВ.ВВОД;

b[i] ← a

конец делать

КОНЕЦ