Массивы

Общая характеристика

Изучение структур данных мы начнём с массивов, т.к. именно массив представляет собой самую популярную структуру данных смежного (то есть размещённую в памяти одним непрерывным блоком) типа. К несомненным достоинствам массивов можно отнести следующие пункты:

- Быстрый доступ к элементам: так как все элементы массива имеют одинаковый размер, можно, зная индекс элемента, мгновенно определить адрес этого элемента в памяти.
- Эффективное использование памяти: тот факт, что все элементы массива имеют одинаковый размер, избавляет от необходимости использовать указатели или специальные разделители между элементами. Получается, что массив не хранит никакой вспомогательной информации (кроме, возможно, собственной длины).
- Локальность в памяти. На первый взгляд это не является преимуществом, но важно помнить, что одна из самых распространённых операций с массивами – это обработка всех элементов в цикле. В современных компьютерах локальность в памяти позволяет использовать эффективные механизмы опережающей выборки и кеш-памяти.

Массивы как структура данных не лишены недостатков. Основным недостатком является невозможность динамически изменять размер массива. Если в программе был объявлен массив из N элементов, то попытка обратиться к $N + k$ (где $k \geq 1$) элементу приведёт к ошибке.

Одним из популярных способов решения этой проблемы является объявление массивов очень больших размеров. Во многих ситуациях этот метод помогает, но его использование приводит к нерациональному расходу памяти. Если в программе используется много массивов, то перерасход может существенно повысить требования к аппаратным характеристикам компьютера.

Вторым по популярности способом решения проблемы постоянства размера массива является использование так называемых динамических массивов. Если возникает необходимость использовать больше элементов, чем позволяет объявленный массив, можно пойти следующим путём: выделить память под новый массив достаточного размера, скопировать в новый массив содержимое исходного массива и освободить память, занятую исходным массивом. В этом методе есть два узких места. Во-первых, во время копирования возникает ситуация, когда в памяти хранятся две копии массива (в старом массиве и в новом), для очень больших массивов такая ситуация может быть неприемлема из-за нехватки памяти. Во-вторых, весьма затратной выглядит процедура копирования массивов, но на самом деле довольно легко можно показать, что копирование не увеличивает асимптотическую сложность.

Предположим, что каждый раз, когда будет возникать ситуация нехватки памяти, будет создаваться массив, вдвое превышающий исходный массив. Начнём рассмотрение с массива из одного элемента и будем добавлять в массив числа $1,2,3,4,$ ….

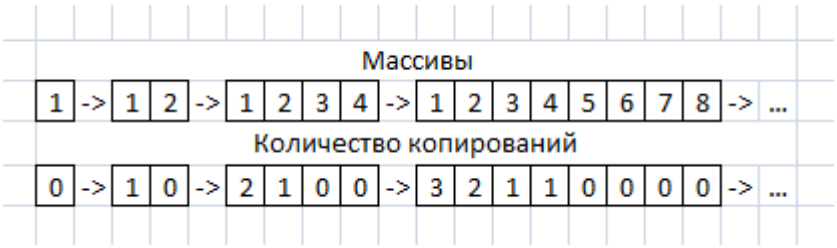

Рис. 1. Пояснение процедуры расширения массива

Очевидно, что для того, чтобы расширить массив до N элементов, необходимо выполнить $\log_2 N$ расширений массива. Также очевидно, что далеко не все элементы будут копироваться при расширении массива (самые первые элементы будут скопированы максимальное число раз). Легко показать, что общее количество копирований оценивается функцией $O(N)$. Т.о. сложность работы с динамическим массивом оценивается функцией $O(N)$ в худшем случае.

Самым большим недостатком в работе с динамическими массивами является понижение производительности при добавлении элементов в наихудшем случае. Подавляющее большинство обращений по-прежнему будет выполняться за константное время, но также изредка будут возникать обращения, приводящие к операциям расширения массива и соответствующим задержкам.

Процедуры и функции для работы с массивами

Рассматривая процедуры и функции, предназначенные для работы с массивами, мы будем предполагать, что у нас уже объявлен достаточно большой[1] статический массив целых чисел A, количество элементов, реально хранящихся в массиве, содержится в переменной N. Для работы с массивами необходимы три основных процедуры:

- `Insert` - вставка нового элемента в массив.
- `Search` - поиск элемента в массиве.
- `Delete` - удаление элемента из массива.

Важно заметить, что реализация этих процедур значительно отличается при использовании упорядоченных и неупорядоченных массивов. Поэтому эти процедуры мы будем рассматривать по отдельности.

Неупорядоченные массивы

Вставка элемента (Insert)

Вставка элемента в неупорядоченный массив – одна из самых простых процедур. На вход процедуры поступает единственная переменная `val` - элемент, который необходимо добавить.

Листинг 1 Вставка элемента в неупорядоченный массив

```
procedure Insert(val: word);
begin
  N:=N+1;
```

[1] "Достаточно большой" в данном контексте означает, что при использовании процедуры вставки элемента не возникнет выхода за границы массива.

```
  A[N]:=val;
end;
```

Очевидно, что сложность процедуры никак не зависит от N, а значит, сложность процедуры `Insert` оценивается как $O(C)$.

Поиск элемента в массиве(Search)

На вход функции поиска поступает единственное значение `val` - элемент, который необходимо найти. Если элемент присутствует в массиве, функция вернёт индекс первого вхождения[2] элемента в массив, если элемента в массиве нет, функция вернёт 0.

Листинг 2 Поиск элемента в неупорядоченном массиве

```
function Search(val: word): word;
var
i: word;
fl: boolean;
begin
  i:=1; fl:=false;
  while (i<=N) and (not fl) do
  begin
    if A[i]=val then
      fl:=true
    else
      i:=i+1;
  end;
  if not fl then
    i:=0;
  Search:=i;
end;
```

В лучшем случае искомый элемент найдётся с первого обращения к массиву, в худшем случае придётся обратиться ко всем N элементам. Т.о. в среднем происходит $N/2$ сравнений, т.е. сложность алгоритма оценивается, как $O(N)$.

Удаление элемента из массива (Delete)

На вход процедуры удаления поступает единственное значение `val` - элемент, который необходимо удалить. Чтобы удалить элемент из массива, его необходимо сначала найти. Так же, как и в случае с поиском, процедура `Delete` будет удалять только первое вхождение элемента в массив. Если необходимо удалить все вхождения, то есть смысл выполнять процедуру `Delete` в цикле, условием выхода из которого является отсутствие удаляемого элемента в массиве.

При удалении элемента подразумевается, что массив не должен содержать пустых элементов. Присутствие дыр усложняет все операции с массивом, потому что перед любым действием с ячейкой её надо проверять на пустоту. Чтобы избежать этих неприятностей, процедура удаления должна возвращать массив без пустых мест. Добиться этого можно, если после удаления элемента все последующие элементы сдвинуть на позицию влево. В итоге общая сложность алгоритма складывается из двух частей: этап поиска и этап сдвига. Оба этапа имеют сложность $O(N)$, а значит, и вся процедура имеет такую же сложность.

---

[2] Во многих случаях этого достаточно, но если необходимо найти все вхождения, то функция Search должна быть переписана с учётом этого требования.

Листинг 3 Удаление элемента из неупорядоченного массива

```
procedure Delete(val: word);
var
i,ind: word;
begin
  ind:=Search(val);
  if ind<>0 then
  begin
    for i:=ind+1 to N do
      A[i-1]:=A[i];
    N:=N-1;
  end;
end;
```

Рис. 2. Пояснение процедуры удаления элемента из массива

Заметим, что в данной реализации ни один элемент не удаляется. На место удалённого элемента копируются элементы, стоящие справа, а последний элемент массива пропадает, т.к. находится за логической границей массива N, а значит, ни одной стандартной процедурой обратиться к нему не получится.

Упорядоченные массивы

Упорядоченные массивы характеризуются тем, что для любых $1 \leq i \leq N-1$ выполняется правило $A[i] \leq A[i+1]$, т.е. в каждой последующей ячейке хранится значение не меньше, чем в предыдущей.

Вставка элемента (OrdInsert)

Требование упорядоченности массива делает процедуру вставки значительно сложнее. Есть много способов реализовать корректное добавление элемента в отсортированный массив. Один из способов заключается в том, чтобы сначала найти позицию, которую займёт добавляемый элемент в отсортированном массиве, потом сдвинуть все элементы к концу массива, чтобы освободить место, и, наконец, вставить элемент. Другой способ заключается в добавлении элемента в конец массива и выполнении одной итерации обменной (пузырьковой) сортировки. В обоих случаях сложность алгоритма равна $O(N)$. В листинге приведена реализация второго метода.

Листинг 4 Вставка элемента в упорядоченный массив

```
procedure OrdInsert(val: word);
var
i: word;
tmp: word;
begin
  N:=N+1;
  A[N]:=val;
  i:=N;
  while (i>=2) and (A[i-1]>A[i]) do
  begin
```

```
    tmp:=A[i-1];
    A[i-1]:=A[i];
    A[i]:=tmp;
    i:=i-1;
  end;
end;
```

Поиск элемента в массиве (BinarySearch)

Свойство упорядоченности приводит к усложнению процедуры вставки, но, в то же время, позволяет существенно ускорить процедуру поиска. Вместо простого линейного просмотра можно воспользоваться двоичным поиском. Алгоритм двоичного (или бинарного) поиска сравнивает элемент в середине исследуемой части массива с искомым. Если искомый элемент меньше, алгоритм продолжает перебирать первую половину исследуемой части массива, если же он больше, чем найденный элемент, поиск продолжается во второй половине массива. Очевидно, что сложность такого алгоритма равна $O(\log_2 N)$.

Листинг 5 Поиск элемента в упорядоченном массиве

```
function BinarySearch(val: word): word;
var
min,max,middle: word;
fl: boolean;
begin
  fl:=false;
  min:=1; max:=N;
  while (min<=max) and (not fl) do
  begin
    middle:=round((max+min)/2);
    if A[middle]=val then
      fl:=true
    else
      if val<A[middle] then
        max:=middle-1
      else
        min:=middle+1;
  end;
  if not fl then
    middle:=0;
  BinarySearch:=middle;
end;
```

Удаление элемента из массива (OrdDelete)

Процедура удаления претерпела наименьшие изменения. Единственное отличие в том, что для поиска удаляемого элемента вместо процедуры Search используется процедура BinarySearch. Важно понимать, что, хоть сложность этапа поиска и уменьшилась, но общая сложность процедуры осталась прежней - $O(N)$.

Листинг 6 Удаление элемента из упорядоченного массива

```
procedure OrdDelete(val: word);
var
i,ind: word;
begin
  ind:=BinarySearch(val);
  if ind<>0 then
```

```
  begin
    for i:=ind+1 to N do
      A[i-1]:=A[i];
    N:=N-1;
  end;
end;
```

Эффективность различных видов организации массивов

На основе всего вышесказанного можно построить следующую таблицу, показывающую эффективность стандартных процедур работы с массивами.

Процедура	Неупорядоченный массив	Упорядоченный массив
Вставка	$O(C)$	$O(N)$
Поиск	$O(N)$	$O(\log_2 N)$
Удаление	$O(N)$	$O(N)$

На основе этой таблицы нельзя дать однозначных рекомендаций по использованию какой-либо реализации массивов. Если в программе происходит разовое добавление большого количества элементов в массив, а в дальнейшем из массива происходит только выборка данных, можно посоветовать следующий подход. Воспользоваться процедурой добавления элементов в неупорядоченный массив, после добавления всех элементов выполнить сортировку эффективным алгоритмом и в дальнейшем пользоваться бинарным поиском.

Специальные виды массивов

Многие программы используют весьма специфичные типы массивов, которые напрямую не поддерживаются языками программирования. К таким "особенным" массивам в первую очередь относятся треугольные и неравномерные массивы.

Треугольные массивы

При реализации многих задач возникает необходимость работать с графами. Один из способов представления графов - это матрица смежности (элемент матрицы $A[i,j]=1$, если между вершинами i,j есть ребро). В неориентированных графах эта матрица симметрична относительно главной диагонали.

Если хранить эту матрицу в виде двумерного массива, то половина элементов будет дублировать значения элементов другой половины. Для небольших массивов такие затраты памяти не слишком важны, но при увеличении количества вершин растут и накладные расходы.

Чтобы сократить затраты памяти, хранят только одну половину матрицы (лежащую выше или ниже главной диагонали). Полученные при этом треугольные массивы часто сворачивают в одномерные массивы следующим образом:

Исходный массив						
#/#	0	1	2	3	4	5
0	A[0,0]	A[0,1]	A[0,2]	A[0,3]	A[0,4]	A[0,5]
1	A[1,0]	A[1,1]	A[1,2]	A[1,3]	A[1,4]	A[1,5]
2	A[2,0]	A[2,1]	A[2,2]	A[2,3]	A[2,4]	A[2,5]
3	A[3,0]	A[3,1]	A[3,2]	A[3,3]	A[3,4]	A[3,5]
4	A[4,0]	A[4,1]	A[4,2]	A[4,3]	A[4,4]	A[4,5]
5	A[5,0]	A[5,1]	A[5,2]	A[5,3]	A[5,4]	A[5,5]
Одномерный массив						
0	A[1,0]	A[2,0]	A[2,1]	A[3,0]	A[3,1]	A[3,2]

Рис. 3. Представление треугольного массива в виде одномерного

Индекс элемента в развёрнутом массиве (ind) вычисляется на основе индексов в треугольном массиве (i, j) по следующей формуле: $ind := round\left(i * \frac{i-1}{2}\right) + j$, для всех $i > j$. Важно помнить, что формулы верны, если индексация массивов начинается с 0[3].

Если в треугольном массиве дополнительно надо учитывать диагональные элементы, то перед использованием формулы необходимо увеличить i на 1.

Нерегулярные массивы

Существуют предметные области, где часто приходится работать с массивами у которых в каждой строке различное количество элементов. Такие массивы называют нерегулярными или рваными. Пример нерегулярного массива представлен ниже:

1				
3	2			
5	1	2	3	7
2	7	8		
9	0	3	1	5

Рис. 4. Пример нерегулярного массива

Для работы с такими массивами можно использовать двумерный массив с количеством столбцов достаточным для хранения любой строки. Если для хранения массива из примера мы бы использовали массив размерностью $5 * 5$, то 9 ячеек этого массива использовались бы впустую. Перерасхода памяти можно избежать, используя уже знакомую идею отображения двумерного массива в одномерный.

[3] При индексации массивов с 1, формулы будут чуть сложнее.

Рис. 5. Представление рваного массива в виде пары одномерных массивов

В данном методе создаётся вспомогательный массив ссылок (назовём его L) и одномерный развёрнутый массив (назовём его B). Важно отметить, что в конец массива L удобно добавить ссылку, указывающую на несуществующий элемент сразу за концом развёрнутого массива, эта ссылка упростит вывод элементов, относящихся к последней строке нерегулярного массива.

Чтобы вывести i-ую строку неравномерного массива, можно воспользоваться следующим кодом:

Листинг 7 Вывод на экран рваного массива

```
for k:=L[i] to L[i+1]-1 do
  write(B[k],' ');
```

Главное преимущество рассматриваемого метода - экономия памяти, а главным недостатком является сложность редактирования. Чтобы добавить элемент в первую строку массива, необходимо сдвинуть в конец все элементы массива B, стоящие правее последнего элемента первой строки, затем нужно добавить единицу ко всем элементам массива L, начиная со второго, и только после этого можно добавлять новый элемент. В рассматриваемом примере будет затронут 21 элемент, при том, что всего хранится 22 элемента. Такие же трудности возникают и при удалении элементов.