Михаил Густокашин. Стеки, очереди, деки, списки.

1 Стеки

Стеком называется структура данных, в которой данные, записанные первыми, извлекаются последними (FILO: First In - Last Out). Например, если мы записали в стек числа 1, 2, 3, то при последующем извлечении получим 3, 2, 1.

Удобно представить стек в виде узкого колодца или рюкзака, в который мы можем класть предмет только наверх и забирать только верхний предмет.

Мы будем реализовывать стек на одномерном массиве, а указателем на вершину стека (первый свободный элемент в массиве) в таком случае будет целочисленная переменная — индекс свободного элемента. Для стека определены две операции `push(x)` — записать в стек элемент (в нашем случае — число) и `pop()` — извлечь из стека элемент.

Размер стека, как и обычно, определим в виде константы:

`#define MAXN 1000`

Сам стек будем описывать в виде структуры.

```
typedef struct
{
  int sp;
  int val[MAXN];
} stack;
```

Мы можем создавать стеки, просто написав, например, `stack a, b;`.

При передаче параметров функции нам нужно будет указывать, с каким конкретно стеком мы хотим работать, а чтобы данные не копировались, будем передавать их по указателю.

```
void push(stack *s, int x)
{
  s->val[s->sp++] = x;
}

int pop(stack *s)
{
  return s->val[--s->sp];
}
```

В функции `push` мы записываем добавляемый элемент в первую свободную позицию, а затем увеличиваем ее номер (постинкремент). В функции pop мы уменьшаем указатель на вершину стека (предекремент), а затем возвращаем значение из последней занятой ячейки.

Для стеков созданных в статической памяти (так, как мы создавали их в примере), вызовы функций будут выглядеть как `push(&a, x); x = pop(&a);` где `x` — число, а `a` — стек. Перед этим необходимо инициализировать указатель на вершину стека нулем (`s.sp = 0`).

Сложность обеих операций над стеком составляет $O(1)$.

Если нам будут необходимы какие-то дополнительные функции работы со стеком, (например, определение пуст ли стек или количества элементов в нем), то всю необходимую информацию мы можем найти в поле `sp`. Напомним, что `sp` — текущее количество элементов в стеке.

При планировании размера стека надо учитывать не общее количество элементов, а максимальное количество одновременно находящихся в стеке элементов (хотя часто эти значения совпадают).

Стеки используются достаточно часто и в большинстве архитектур компьютеров реализованы аппаратно. Одно из применений стеков мы рассмотрим ниже.

2 Очереди

Очередь имеет интуитивно понятное название. Элемент, который попал в очередь раньше, выйдет из нее также раньше (т.е. элементы извлекаются в порядке поступления). По-английски очередь называется queue («кью») или FIFO (First In - First Out).

Очередь мы будем реализовывать на одномерном массиве, аналогично стеку. Тут нам придется немного отойти от аналогий с реальностью для повышения производительности. Если реализовывать очередь в программе как очередь в магазине, где люди постепенно двигаются к кассе, то извлечение элемента из очереди будет иметь сложность $O(N)$, где N - количество элементов в очереди), т.к. все элементы будет необходимо передвинуть. Гораздо удобнее в нашей ситуации перемещать «кассу», т.е. изменять указатель на начало очереди.

Однако в этом случае возникает другая проблема: если в предыдущем случае размер очереди ограничивался количеством одновременно находящихся в ней элементов, то здесь нам необходимо будет создавать очередь с размером, равным общему количеству элементов, которые в ней побывают.

Эту проблему мы решим, закольцевав очередь. Можно представить круг, где помечены две позиции — с какой уходить, и на какую становиться новому элементу. Для этого в реализации мы будем брать оба указателя по модулю $MAXN$.

Очередь также реализуем в виде структуры:

```
typedef struct
{
  int qh, qt;
  int val[MAXN];
} queue;
```

Теперь мы можем создавать очереди, просто написав `queue a, b;`/

Для очереди существует две операции: извлечь элемент из головы (head) очереди (`deq`) и добавить элемент в хвост (tail) очереди (`enq`).

```
void enq(queue *q, int x)
{
  q->val[(q->qt++)%MAXN] = x;
}

int deq(queue *q)
{
  return q->val[(q->qh++)%MAXN];
}
```

Как и в прошлый раз, необходимо передавать в функции указатель, т.е. их вызов должен выглядеть как: `enq(&a, x); x = deq(&a);` где `x` — число, а `a` — очередь. Так же, как и в случае со стеком, мы должны предварительно инициализировать оба указателя очереди нулями: `a.qt = 0; a.qh = 0;`

Признаком того, что очередь пуста или переполнилась, следует считать равенство полей `qt` и `qh`. Количество элементов в очереди определяется так: `qlen = (qt-qh)%MAXN`.

Очереди часто используются в качестве буферов и во многих устройствах реализованы аппаратно.

3 Деки

Деком (deque) называется структура, в которой добавление и извлечение элементов возможно с двух сторон. Т.е. это некоторая смесь стека и очереди.

Для дека можно использовать абсолютно ту же структуру, что для очереди, но функций будет уже 4 (добавление и извлечение в начало и в конец). Для такой структуры данных мы приведем просто исходный текст, все делается полностью аналогично стекам и очередям.

```
typedef struct
{
  int dh, dt;
  int val[MAXN];
} deque;

void push_front(deque *d, int x)
{
  if (d->dh < 1) d->dh += MAXN;
  d->val[(--d->dh)%MAXN] = x;
}

void push_back(deque *d, int x)
{
  d->val[(d->dt++)%MAXN] = x;
}

int pop_front(deque *d)
{
```

```
  return d->val[(d->dh++)%MAXN];
}

int pop_back(deque *d)
{
  if (d->dt < 1) d->dt += MAXN;
return d->val[(--d->dt)%MAXN];
}
```

Единственное усложнение состоит в том, что мы добавили проверку на то, чтобы указатели не становились отрицательными. Тогда определение количества элементов в деке будет выглядеть следующим образом: `dlen = a.dt>a.dh` что эквиваленно записи:

```
if (a.dt > a.dh) dlen = (a.dt-a.dh)%MAXN;
else dlen = MAXN - (a.dh-a.dt)%MAXN;
```

Здесь `a` — дек. Напомним, что перед использованием дека следует установить в ноль поля `dh` и `dt`.

4 Динамически расширяемые массивы

До сих пор мы использовали статические массивы, их размер был определен заранее и не мог изменяться в процессе работы программы. Это обычный и правильный метод, отходить от которого стоит только в некоторых ситуациях. Например, когда мы имеем много массивов, которые могут быть произвольной длины, и известно только ограничение на сумму их длин или длинная арифметика (в этом случае необходимо хранить цифры в обратном порядке, прижатыми к левому краю, а не в прямом порядке, как об этом говорилось в первой лекции).

Вначале мы выделяем какое-то количество памяти под массив, а затем, по мере необходимости, расширяем массив. Если пользоваться наивным методом, т.е. увеличивать массив на 1 элемент каждый раз, когда мы вышли за текущий размер, то производительность будет очень мала. Операция расширения массива требует некоторых накладных расходов, связанных с работой ОС по перераспределению памяти, а в неудачном случае требуется еще и копирование всего содержимого массива в новую область памяти. Это связано с тем, что массив в языке Си обязан быть непрерывным, а расширить его на существующей памяти не всегда возможно.

Мы будем реализовывать компромиссный по времени и требуемой памяти вариант, он будет выполнять $\log N$ выделений памяти (где N — количество элементов в массиве), а неиспользованной останется не больше половины памяти, выделенной под массив.

Вначале мы создадим массив некоторого начального размера, а когда количество использованных элементов будет приближаться к текущему размеру — будем увеличивать размер массива вдвое. Это и даст нам требуемую сложность в $\log N$ выделений памяти, а половина неиспользованной памяти возникнет в случае, если мы прекратили добавление элементов сразу после очередного расширения массива.

Для использования функций выделения памяти в языке **Си** мы должны подключить библиотеку `stdlib.h`.

Допустим, перед нами стоит задача считать неизвестное заранее количество чисел до конца файла. Это можно реализовать с помощью следующего фрагмента программы:

```
int now=0, size=2, i, *a;
a=(int*)malloc(sizeof(int)*size);
while (scanf("%d", &a[now++]) == 1)
  if (now >= size-1) {
    size*=2;
    a=(int*)realloc(a, sizeof(int)*size);
  }
```

В конце программы (или когда массив перестал быть нам нужным), необходимо освободить выделенную память. Это делается с помощью функции `free(a);`

В общем случае, динамически расширяемые массивы являются довольно неплохим средством хранения неизвестного заранее количества данных с относительно небольшими накладными расходами. В динамически расширяемых массивах можно делать все то же самое, что и в обычных массивах, это делает их весьма привлекательными.

Кроме того, по необходимости, размер массива можно уменьшать той же самой функцией `realloc`; можно уменьшать размер массива вдвое, если из него происходит удаление (логично делать это в случае, если `now*2 < size`). Однако, чтобы избежать слишком частого изменения размеров массива, когда он почти заполнен, можно поставить условие на уменьшение `now*4 < size`, но размер массива по-прежнему уменьшать вдвое.

5 Списки

Рассмотрим еще одну динамическую структуру данных, называемую связным списком. Каждый элемент списка представляет собой структуру, одно поле которой содержит информацию (ключ), а другое — ссылку на следующий элемент. Существуют также двусвязные списки, в которых хранится ссылка не только на следующий элемент, но и на предыдущий. Начинается список с указателя на элемент списка. В целом его вид можно представить следующим образом:

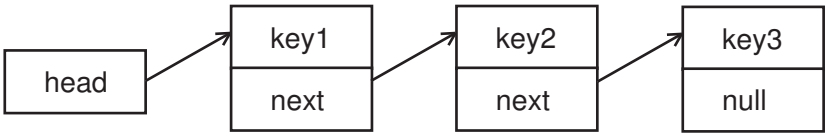

Рис. 1: Общий вид списка

Один элемент списка будем описывать с помощью следующей структуры:

```
typedef struct
{
  int key;
  struct _list *next;
} list;
```

Чтобы работать со списком, необходимы указатели на элемент, которые заводятся следующим образом: `list *head=NULL, *temp;` Довольно необычно происходит операция добавления в список: чтобы добиться сложности $O(1)$, добавление происходит в начало списка. Это реализуется следующим фрагментом кода:

```
  temp = (list*)malloc(sizeof(list));
  temp->key = newkey;
  temp->next = head;
  head = temp;
```

Первая строка выделяет память под новый элемент, вторая записывает новое значение ключа, третья присваивает полю `next` вновь созданного элемента указаетль на текущее начало списка, а четвертая устанавливает указатель на начало списка на вновь созданный элемент. Порядок действий показан на рисунке:

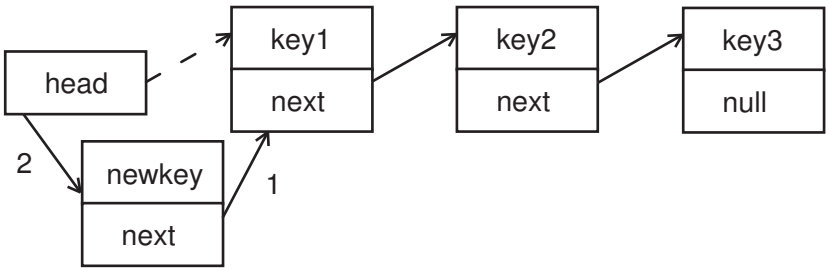

Рис. 2: Добавление элемента в начало списка

Аналогично осуществляется вставка после элемента, на который имеется ссылка, достаточно заменить `head` в нашем коде на его поле `next`.

Поиск элемента по ключу осуществляется за $O(N)$ — нам необходимо пройти весь список. Напишем функцию, которая возвращает указатель на элемент по его значению ключа или `NULL`, если элемента с таким ключом не существует.

```
list* find(list* head, int key)
{
  list *now=head;
  while (now != NULL) {
    if (key == now->key) break;
    now = now->next;
  }
  return now;
}
```

Однако, удаление элемента невозможно реализовать только зная ссылку на него (т.к. необходимо, чтобы список остался связным, а удаление элемента создаст в нем «дырку»). Напишем отдельную функцию удаления элемента с заданным ключом и возвращающую ссылку на новый список (без этого элемента).

```
list* del(list *head, int key)
{
  list *now=head, *prev;
  if (key == head->key) {
    head = head->next;
    free(now);
  } else {
    prev = now;
    now = now->next;
    if (key == now->key) {
      prev->next = now->next;
      free(now);
    }
  }
  return head;
}
```

Здесь мы отдельно рассматриваем случай, когда необходимо удалить первый элемент списка, т.к. он не имеет предыдущего. В противном случае мы делаем удаление согласно рисунку:

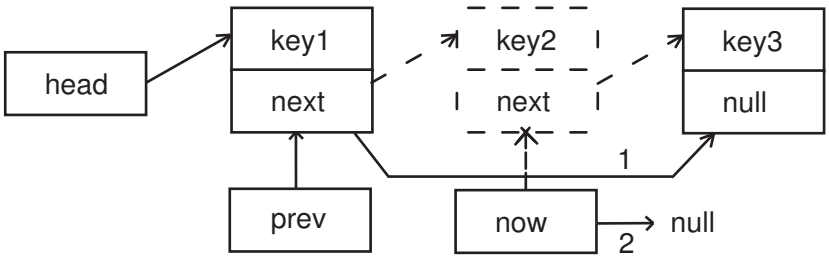

Рис. 3: Удаление элемента из списка

Сначала осуществляем присваивание поля `next` для предыдущего элемента (действие 1), а затем удаление текущего элемента (действие 2).

Над списком можно определить множество различных функций (например, объединение двух списков, удаление элементов, обладающих каким-либо признаком и т.д.), но они все базируются на изложенных выше идеях.

6 Сравнение динамических структур

Мы изучили две структуры в динамической памяти: динамически расширяемый массив (вообще говоря, массив изменяемого размера, т.к. он может и уменьшаться) и списки. Во-первых, оценим требования к памяти. Динамически расширяемый массив может иметь такое же количество пустых элементов, как и непустых, т.е. его требование к памяти, в худшем случае, записывается как `2*N*sizeof(key)`, для списка это требование записывается в общем случае как `N*(sizeof(key)+sizeof(*key))`. Здесь `N` — количество элементов в структуре, `sizeof(key)` — размер ключа, `sizeof(*key)` — размер указателя (обычно 4 байта). Так для ключа типа `int` (4 байта) худший случай динамически расширяемого массива совпадает с обычным случаем списка.

Производительность по времени работы измерялась в тиках (`GetTickCount`) на 10^7 элементах:

	Создание	**Поиск**	**Удаление**	**Доступ**
Массив	422	31	78	0
Список	4688	63	63	63

В обоих случаях структуры заполнялись последовательными числами от 0 до $10^7 - 1$, поиск, удаление и доступ по индексу осуществлялись к элементу с номером $10^7/2$. Для заполнения, поиска и удаления элемента в списке использовались приведенные выше функции, при удалении элемента в середине массива осуществлялся сдвиг конца массива («дырка» не образовывалась).

Оставим значения в таблице без комментариев, и в дальнейшем будем использовать списки в задачах, где критична производительность (большое количество элементов в списках) только в случае крайней необходимости.

Крайняя необходимость возникает, когда происходит много вставок (или удалений) в средину и, особенно, если позиция для вставки (удаления) не сильно отличается от текущего положения.

7 Выделение памяти на массиве

Начнем с рассмотрения «алгоритма», с помощью которого можно реализовывать динамические структуры на статическом массиве. Это позволит нам понять, как операционная система выделяет память по запросам, что в некоторых ситуациях поможет заметно ускорить выполнение программы. Не секрет, что выделение и очистка памяти (`malloc` и `free`) занимают довольно значительное время, которое тратится на вызов и работу функций ОС.

Будем реализовывать выделение и очистку памяти на массиве с помощью односвязного списка свободных ячеек. В качестве указателя у нас будет выступать индекс ячейки. Будем хранить «указатель» на первую свободную ячейку в переменной `ffree` (first free — первый свободный элемент), сам массив будет состоять из ячеек такого типа:

```
typedef struct {
<type> data;
int next;
} val;
```

Здесь `<type>` — тип переменной, в которой хранится осмысленное значение (`data`), а `next` — служебное поле, указывающее на следующий свободный элемент. Практически всегда можно обойтись без дополнительного поля `next`, т.к. такой тип организации используется для динамических структур (например, если мы хотим хранить список, который в свою очередь имеет «указатель» на следующий элемент, то дублировать этот указатель нет смысла).

Перед тем как приступить к работе следует для каждого индекса `i` установить поле `next` в `i+1`, а переменную `ffree` в 0. Это будет означать, что весь массив представляет собой пустой список, а первая свободная ячейка имеет индекс 0.

Пусть мы хотим выделить новый участок памяти в массиве mem и установить на него указатель `it`. Это будет выглядеть так:

```
it = ffree;
ffree = mem[ffree].next;
mem[it].next = -1;
```

Мы перемещаем указатель на начало списка, на следующий свободный элемент из списка. В принципе, поле `next` для указателя `it` можно не менять или устанавливать осмысленное значение. Теперь рассмотрим удаление элемента по указателю `it`:

```
mem[it].next = ffree;
ffree = it;
```

Здесь все тоже более или менее понятно: освободившуюся ячейку мы записываем в начало списка свободных ячеек. Примерно так и работают алгоритмы выделения памяти в ОС, хотя, конечно, там используются более изощренные методы, т.к. ОС умеет выделять участки памяти разной длины и хранит для них эффективные индексы, в случае необходимости осуществляет перестановки и копирование в памяти и проч.

Такой метод дает некоторый выигрыш в скорости и его надо использовать в тех случаях, когда вы эффективно реализуете неэффективный алгоритм в надежде обхитрить жюри, подсунув им алгоритмически неэффективное решение, укладывающееся в TL. Также мы имеем больше информации о взаимном расположении элементов в памяти, что в некоторых ситуациях позволит сильно ускорить выполнение операций.

Обратите внимание, что реализация подобного «динамического выделения памяти» на динамически расширяемом массиве не имеет особого смысла, т.к. при этом теряется эффективность (теряется время на перевыделения памяти).