Центр дополнительного образования детей
Дистантное обучение

117630, Москва, ул. акад. Челомея, д. 8б, тел./факс 936-3104

Курс «Олимпиадные задачи по программированию».
Преподаватель: Михаил Сергеевич Густокашин

Лекция 2
Битовые операции и структуры данных (часть 1).

Оглавление:

msg@list.ru
http://g6prog.narod.ru
http://desc.ru

Битовые операции.

Как известно, в компьютерах обычно применяется двоичное представление числа. Сейчас нас не будет интересовать представление числа в архитектуре x86, мы будем считать, что число записывается так, как это принято: слева стоят старшие разряды, справа – младшие.

Мы изучим следующие операции, комбинациями которых будем добиваться нужных результатов: отрицание (замена всех 0 на 1 и наоборот), или, и, исключающее или (поразрядное сложение по модулю 2). Операции записываются, соответственно, как `~`, `|`, `&`, `^` (в Паскале `not`, `or`, `and`, `xor`, написание совпадает с написанием аналогичных логических операций, а что именно имелось в виду, определяется из контекста).

Кроме того, существует еще две операции, которые нам пригодятся: сдвиг влево и сдвиг вправо (с дополнением нулями справа и слева соответственно). В Си они обозначаются, как `<<` и `>>`, в Паскале: `shl` и `shr`.

Рассмотрим несколько примеров.

1) Установить k-ый справа бит числа `n` в 1. Разделим операцию на два этапа: создание числа с единственной 1 на k-ой позиции и логическое или с переменной `n`.

```
j = 1;
j <<= k;
n |= j;
```

В битовых операциях также можно использовать сокращенную запись операций. На Паскале это будет записываться так:

```
j := 1;
j := j shl k;
n := n or j;
```

2) Проверить, является ли k-ый бит переменной n единицей. Здесь все делается почти также.

```
j = 1;
j <<= k;
j &= n;
```

Если j отлично от 0, то k-ый бит был 1, иначе 0.

3) Проверить, есть ли в двоичной записи числа n хотя бы один 0. Идея заключается в следующем: создадим переменную, полностью состоящую из единиц, а затем сравним с n.

```
j = 0;
j = ~j;
```

Если n и j совпадают, то ни одной единицы в записи n нет.

4) Установить k правых бит переменной n в нули. Для решения этой задачи воспользуемся применением операций сдвига вправо (при этом самые правые биты удалятся) и сдвига влево на то же количество элементов (эти позиции заполнятся нулями).

```
n <<= k;
n >>= k;
```

5) Дано n чисел, каждое из которых встречается в последовательности два или кратное двум число раз, кроме одного, которое встречается нечетное число раз. Найти это число. Для решения этой задачи следует воспользоваться следующими утверждениями: `x^x = 0`, $\forall x$ (читается как «x исключающее или x равно 0 для любого x», также это может быть записано как $x \wedge x \equiv 0$ - «x исключающее или x тождественно равно 0») и $(x \wedge y) \wedge x \equiv y$. Исходя из этих соображений можно просто применить исключающее или («поксорить») все числа, и в результате останется искомое число, т.к. встречающееся четное количество раз числа сократятся.

```
k = 0;
For (i, n)
```

```
{
      scanf("%d", &x);
      k ^= x;
}
```

Довольно частым применением битовых операций являются битовые булевские массивы (они занимают в 8 раз меньше памяти, чем обычные булевские массивы, т.к. для хранения значения «истина» или «ложь» достаточно одного бита).

Определим как константу максимальный размер массива. Мы будем использовать в качестве носителя тип `int`, который состоит из 32 бит (в современных компиляторах).

```
#define MAXN 1000
int bitarr[MAXN];
```

Таким образом, мы получим массив из 32000 бит.

Опишем три функции `set(n)` – установку n-го бита в 1, `unset(n)` – установку n-го бита в 0 и `get(n)`, которая будет возвращать значение n-го бита. Все эти операции мы уже рассматривали (установку бита в 0 и 1 и получение значения бита), поэтому приведем пример только одной из функций, например, get(int n), а остальные несложно сделать по аналогии.

```
1    int get(int n)
2    {
3         int seg, off, j = 1;
4         seg = n / 32;
5         off = n % 32;
6         j <<= off;
7         if ((bitarr[seg] & j) != 0) return 1;
8         else return 0;
9    }
```

Здесь `seg` – индекс элемента в массиве, куда попадет данный бит, а `off` – номер бита в этом элементе.

Следует заметить, что такой метод работает медленнее, чем обычный булевский массив и должен использоваться только в случаях, если нам очень критична память или существует необходимость в какой либо специфичной проверке, например, найти хотя бы один 0 среди большого количества 1.

Стеки.

Стеком называется структура данных, в которой данные, записанные в первую очередь, извлекаются также в первую очередь (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 в конце этой лекции.

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 FILO (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.

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 функции 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 для того, чтобы указатели не становились отрицательными. Тогда определение количества элементов в деке будет выглядеть следующим образом:

```
dlen = a.dt>a.dh?(a.dt-a.dh)%MAXN:MAXN-(a.dh-a.dt)%MAXN;
```

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`.

Кучи.

Куча (по-английски heap) – структура данных, которая может выдавать минимальный (или максимальный) элемент за O(1), добавление нового элемента и удаление минимального элемента происходит за O(logN) (где N – количество элементов в куче). Другие операции над кучей не определены (хотя при необходимости могут быть введены, однако эффективность их будет невысокой, и они обязаны поддерживать свойства кучи).

Другое название кучи – очередь с приоритетами, что и отражает ее сущность.

Сразу перейдем к рассмотрению реализации кучи на одномерном массиве. Назовем элементы с индексами i*2+1 и i*2+2 потомками элемента с индексом i. Элемент i будет называться предком этих элементов. Несложно заметить, что потомки двух разных элементов не пересекаются и каждый элемент, кроме нулевого, является чьим-либо потомком. Основное свойство кучи: каждый элемент не больше своих потомков.

Например, массив 1, 6, 8, 7, 12, 9, 10 может являться кучей (напомним, что индексация в массиве начинается с нуля).

Для хранения кучи создадим структуру, аналогичную предыдущим:

```
typedef struct
{
	int hs;
	int val[MAXN];
} heap;
```

Как было написано выше, опишем три функции. В первую очередь напишем функцию, возвращающую наименьший элемент. Она будет очень простая, т.к. из свойства кучи следует, что минимум находится в нулевом элементе:

```
int get_min(heap *h)
{
	return h->val[0];
}
```

Перед использованием этой функции необходимо обязательно проверить, что куча не пуста!

Следующей определим функцию добавления элемента в кучу. Новый элемент будем добавлять в конец кучи, а затем обменивать его с предком, пока он меньше, чем его предок или мы не достигли нулевого индекса. При этом свойство кучи не нарушится.

```
void add_heap(heap *h, int x)
{
      int y, pos=h->hs, npos;
      h->val[h->hs++] = x;
      npos=(pos-1)/2;
      while (pos && h->val[pos] < h->val[npos])
      {
           y=h->val[pos];
           h->val[pos]=h->val[npos];
           h->val[npos] = y;
           pos=npos;
           npos=(pos-1)/2;
      }
}
```

Как мы писали выше, сложность добавления элемента в кучу составляет O(logN) – каждый раз индекс текущего элемента уменьшается вдвое.

Кроме того, часто возникает задача удаления минимального элемента. Мы будем реализовывать это следующим образом: запишем на место нулевого элемента последний, уменьшим размер кучи на 1 и просеем нулевой элемент по куче так, чтобы сохранилось свойство кучи. Будем реализовывать просеивание следующим образом: если элемент больше, чем меньший из своих потомков, то меняем их местами и продолжаем процесс, пока выполнено условие или мы не вышли за пределы кучи. В реализации этого алгоритма применена одна хитрость, которая будет пояснена ниже.

```
void del_heap(heap *h)
{
     int minp, pos=0, y;
     h->val[0]=h->val[--h->hs];
     while (pos*2+1 < h->hs)
     {
          y=pos*2+1;
          minp=h->val[y]<h->val[y+1]?y:y+1;
          if (h->val[pos]>h->val[minp])
          {
               y=h->val[pos];
               h->val[pos]=h->val[minp];
               h->val[minp]=y;
               pos=minp;
          } else break;
     }
}
```

Алгоритм делает ровно то, что написано выше. На первый взгляд, единственный случай, где теоретически возможна ошибка, когда у нас имеется всего один потомок (т.е. `pos*2+2 == h->hs`) . Такой вариант возможен, если количество элементов в куче четно.

В этом случае мы выбираем минимум из первого потомка и первого элемента вне кучи, что, казалось бы, является грубой ошибкой. Но мы знаем, что просеиваемое число и первое число вне кучи, равны. Это гарантирует нам, что обмена с элементов вне кучи не

произойдет, а если необходим обмен с единственным потомком, то он будет выполнен корректно.

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 языках Си и Паскаль 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 logN 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 logN 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);
     }
```

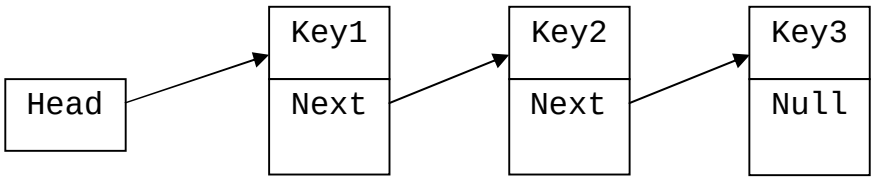
В конце программы (или когда массив перестал быть нам нужным), необходимо освободить выделенную память. Это делается с помощью функции `free(a);`

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

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:



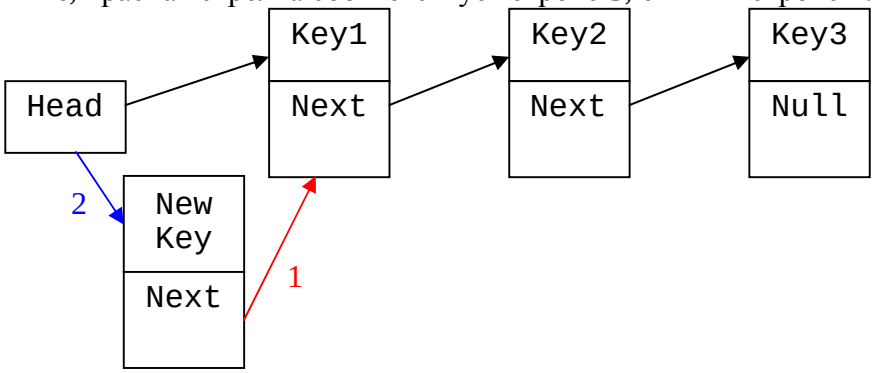
Один элемент списка будем описывать с помощью следующей структуры:

```
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:

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

Первая строка выделяет память под новый элемент, вторая записывает новое значение ключа, остальные поясним рисунками. Черными стрелками помечено старое состояние, красная стрелка соответствует строке 3, синяя – строке 4:



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.

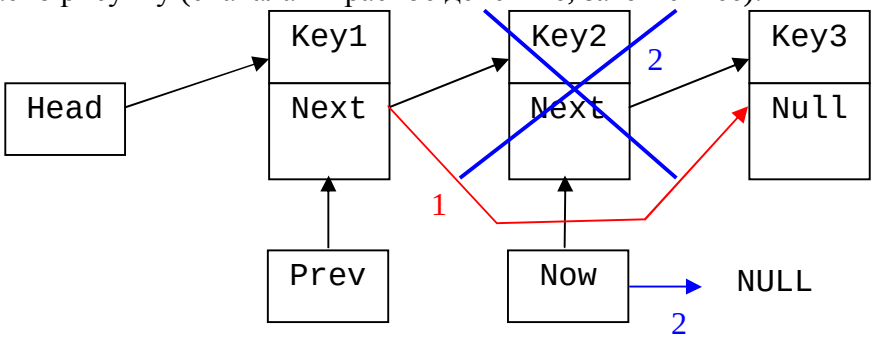
```
list* find(list* head, int key)
{
      list *now=head;
      while (now != NULL)
      {
```

```
            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 (сначала – красное действие, затем синее):



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.

Сравнение производительности 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 элемента в середине	Удаление элемента в середине	Доступ к элементу по индексу
Динамически расширяемый массив	422	31	78	0
Список	4688	63	63	63

В обоих случаях структуры заполнялись последовательными числами от 0 до 10^7, 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 в средину и, особенно, если позиция для вставки не сильно отличается от текущего положения.

Хеш-таблицы.

Допустим, перед нами стоит следующая задача: нам дается множество ключей (уникальных значений) и требуется уметь быстро проверять, входит ли ключ в наше множество. При этом множество ключей может изменяться, т.е. ключи могут добавляться и исключаться из множества.

Сейчас мы будем рассматривать все на числовых примерах. Допустим, нам дан набор целых чисел от 1 до 10000, а затем идет серия запросов вида «есть ли число X в множестве?». Мы можем создать булевский массив, в котором будем помечать, встречалось данное число или нет. При этом сложность одного запроса будет O(1).

Если же чисел больше, то мы можем попробовать использовать для хранения признака наличия числа во множестве не 1 байт, а 1 бит, пользуясь битовыми функциями, описанными в первом разделе лекции. Это даст нам возможность увеличить максимальный размер числа в 8 раз. Но и этого может не хватить. Например, если числа изменяются от 0 до 2^{31}, то такую таблицу невозможно разместить в памяти, которая дается нашему решению.

Эта задача имеет решение в некоторых частных случаях. Например, пусть максимальный размер множества – 1000, а каждый элемент может быть в пределах от 0 до 2^{31}. Если мы будем создавать таблицу размером 2^{31} элементов, то будет использована меньше ее одной миллионной части.

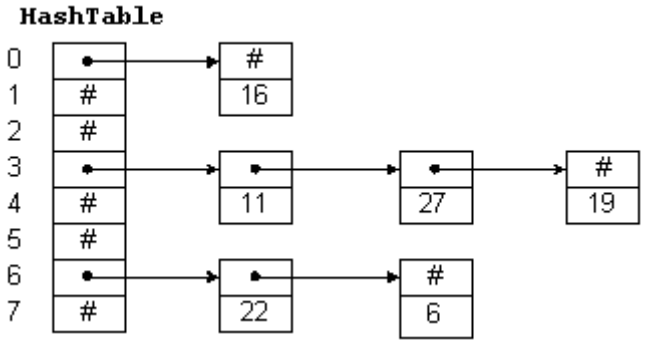
Будем решать такой класс задач (когда количество чисел намного меньше максимального значения) с помощью так называемых хеш-таблиц.

Введем понятие хеш-функции, как функции, отображающей множество ключей (в нашем случае чисел от 0 до 2^{31}) в меньшее множество ключей (соизмеримое с максимальным количеством элементов – в нашем случае с 1000). Хеш-функция должна обладать двумя основными свойствами: быть быстрой и равномерно генерировать ключи (т.е. чтобы одному и тому же ключу в малом множестве соответствовало примерно равное количество ключей в большом множестве). Иногда от хеш-функции требуют неустойчивости (т.е. чтобы при близких значениях ключей большого множества она генерировала различные ключи малого множества).

Размер таблицы (вообще говоря, одномерного массива) для эффективной работы должен быть больше, чем удвоенное количество элементов малого множества. Обозначим размер таблицы за N.

Будем использовать в качестве хеш-функции операцию взятия остатка от деления числа большого множества на N (X % N). Это достаточно хорошая функция: считается относительно быстро и распределена равномерно. Итак, мы считаем остаток от деления нашего числа X на N и записываем X в ячейку с индексом X % N. Затем, при проверке числа Y, мы просто смотрим на ячейку Y % N и, если число Y находится там, то возвращаем признак наличия. Сложность опять получается O(1).

Однако возникает проблема – несколько чисел могут иметь одинаковый остаток от деления на N. Такая ситуация называется «коллизией». Существует несколько способов разрешения коллизий, мы рассмотрим способ со списками, каждый из которых соответствует одному значению остатка (одной ячейке хеш-таблицы).



Допустим, размер нашей хеш-таблицы равен 8 и в нее были внесены элементы 6, 19, 27, 11, 16, 22. Тогда она будет выглядеть как на рисунке.

Использование списков в данном случае уместно, т.к. количество элементов в каждом списке будет небольшим. Можно использовать и динамически расширяемые массивы, если значений в хеш-таблице достаточно много. Теперь для добавления элемента нам надо подсчитать его хеш-функцию и поместить в соответствующий этому значению список. Для проверки принадлежности элемента множеству нам также надо посчитать его хеш-функцию и попытаться найти его в нашем списке.

В среднем такая хеш-таблица будет работать за O(1), т.к. коллизии будут встречаться редко. Ее можно «завалить» по времени только в случае, если точно знать размер таблицы, а при использовании набора тестов для проверки задачи это невозможно. Поэтому следует выбирать произвольный размер, больший 2*N, где N – максимальное количество элементов, одновременно находящихся в хеш-таблице.

Функции работы с хеш-таблицей можно реализовать так:

```
typedef struct {
      struct -node *next;
      int data;
} node;

node **hashTable;
int hashTableSize;

int hash(int data) {
      return (data % hashTableSize);
}

node *insertnode(int data) {
      node *p, *p0;
      int bucket;
      bucket = hash(data);
      p = (node*) malloc(sizeof(node));
      p0 = hashTable[bucket];
      hashTable[bucket] = p;
      p->next = p0;
```

```
        p->data = data;
        return p;
    }

    void deletenode(int data) {
        node *p0, *p;
        int bucket;
        p0 = 0;
        bucket = hash(data);
        p = hashTable[bucket];
        while (p && (p->data != data)) {
             p0 = p;
             p = p->next;
        }
        if (!p) return;
        if (p0)
             p0->next = p->next;
        else
             hashTable[bucket] = p->next;
        free (p);
    }

    node *findnode (int data) {
        node *p;
        p = hashTable[hash(data)];
        while (p && (p->data != data))
             p = p->next;
        return p;
    }
```

Перед работой необходимо создать хеш-таблицу в динамической памяти и установить ее размер. Например:

```
hashTableSize = 20001;
hashTable = (node**) malloc(hashTableSize*sizeof(node*));
```

Кроме того, хеш-функции могут использоваться и в других ситуациях, например при сравнении сложных объектов. Мы заранее считаем хеш-функцию от каждого объекта, а затем, при сравнении, в первую очередь сравниваем значения хеш-функция и проводим сравнение для объектов полностью только в случае совпадения хешей.

Например, простейший способ подсчитать хеш-функцию от строки – сложить коды всех символов, входящих в строку. Такая хеш-функция будет генерировать одинаковые значения для строк, которые содержат одинаковые буквы, независимо от порядка.

Существуют и другие методы подсчета хеш-функции от строки, например подсчет полинома по какому-либо модулю.

Хеш-функция от строки обычно должна хорошо пересчитываться при удалении первого символа строки и добавлении нового символа – это позволяет использовать такое хеширование при поиске подстроки в строке (метод Рабина-Карпа).

В принципе, хеш-функция может быть введена для любых объектов.

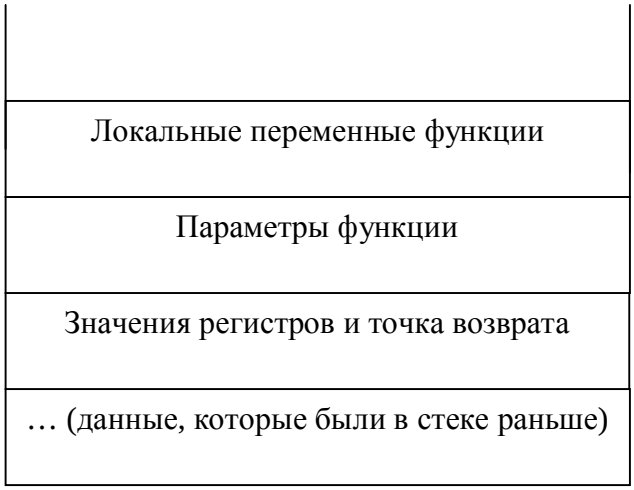
Механизм вызова функций и рекурсивные функции.

Итак, под конец лекции обещанный пример использования стеков. Рассмотрим механизм запуска функции в программе.

При вызове функции мы должны: 1) сохранить текущие значения регистров процессора, 2) запомнить «точку возврата», т.е. то место, куда мы должны вернуться

после выполнения функции (вообще говоря, это тоже специальные регистры процессора), 3) передать параметры в функцию и 4) выделить место под локальные переменные. Сам код функции хранится отдельно в оперативной памяти.

Все эти данные помещаются в стек. Таким образом, он имеет следующий вид:



Как видно, для вызова функции требуется достаточно много накладных расходов, поэтому короткие функции следует оформлять в виде макросов или делать их inline (встраиваемыми), что является, по сути, тем же макросом в «осовремененном» виде.

Перейдем теперь к рекурсивным вызовам функции. Условимся разделять «функцию» (машинный код команд функции) и «экземпляр функции» (совокупность содержимого области стека для этой функции вместе с машинным кодом).

На следующем рисунке красным обозначено содержимое стека для экземпляра функции rec с параметром n=2, синим – с n=1 и фиолетовым с n=0.

При n=0 рекурсивные вызовы прекратятся, мы выведем текущее (фиолетовое) значение n – 0, вернемся в фиолетовую точку возврата rec(1) и вынем все фиолетовое из стека. Теперь мы работаем снова в синей части стека, и находимся на позиции `printf`. Мы выводим синее значение n (1) и возвращаемся в синюю точку возврата rec(2). Теперь мы находимся на позиции `printf` в красном экземпляре функции. Выводим красное значение n (2), и переходим по красной точке возврата в функцию main().

Вывод программы будет 0 1 2.

```
void rec(int n)
{
     if (n > 0)
          rec(n-1);
     printf("%d ", n);
}

int main(void)
{
     rec(2);
     return 0;
}
```