РАЗРАБОТКА ЭФФЕКТИВНОГО С ТОЧКИ ЗРЕНИЯ ИСПОЛЬЗОВАНИЯ КЭШ-ПАМЯТИ СПИСКА СТРУКТУР (операции добавления, удаления, вставки, поиска; размер списка фиксированный)

Для разработки эффективного алгоритма попробуем разместить элементы списка рядом => **Расположим список**

```
struct hdr {
        struct hdr *prev, *next;
};
typedef
struct List {
    int elem;
    struct hdr  h;
}  list_t;                                                  //размер структуры — 12 б
```

в массиве.

Рассмотрим функции выделения памяти и добавления элемента в список (=> вставки), поиска и удаления элемента из списка. В **первом** случае используются обычные malloc и free (в оп-ции delete), третий случай рассмотрим позже, во **втором** случае они реализованы следующим образом:

Функция **NEWMALLOC** при вызове:
Если список «empty» (список, содержащий пустые элементы) пустой, то создаёт двунаправленный список «mas» и передаёт ему указатели таким образом, что «mas» закольцовывается на «empty» (то есть последний элемент списка «empty» указывает на первый элемент «mas», и наоборот);
Если «empty» существует, или для него уже выделена память в предыдущем действии, то создаёт указатель на следующий элемент списка «empty», удаляет этот элемент из «empty» и отправляет его в return функции:

Иллюстрация работы функции NEWMALLOC для данного варианта:

```
int IS_EMPTY(struct hdr *p) {
    if (p->next == p) {
        return 1;
    }
    return 0;
}

list_t* NEWMALLOC(void){
    int i;
    if (IS_EMPTY(&empty)) {                                 // если список свободных
                                                            // элементов пустой =>
        list_t *mas = (list_t*)malloc(sizeof(list_t)*LEN);  // выделяем ему память
        for (i = 0; i < LEN; i ++) {
            mas[i].h.next = &(mas[i + 1].h);
            mas[i].h.prev = &(mas[i - 1].h);
        }
        mas[0].h.prev = &empty;                         // закольцовываем на данный
```

```
        empty.next = &(mas[0].h);                    // список p
        mas[i - 1].h.next = &empty;
        empty.prev = &(mas[i - 1].h);
    }

    struct hdr *buf = empty.next;                    // иначе предоставляем
    empty.next = empty.next->next;                   // элемент из пустого (empty)
    empty.next->prev = empty.next;
    return hdr2elem(buf);
}
```

Функция **DELETE** при вызове:
Удаляет данный элемент из списка структур «p» и добавляет его в список свободных элементов как следующий элемент «empty». Таким образом, мы не оставляем пустых элементов в массиве, в котором лежит список «p».

Иллюстрация работы функции DELETE для данного варианта:

```
int DELETE(list_t* root){
    root->h.prev->next = root->h.next;               // удаляем элемент из списка p
    root->h.next->prev = root->h.prev;
    PUSHAFTER( empty.prev, root );                   // добавляем его в список свободных
    return 1;
}
```

Чтобы объяснить быстродействие функций второго случая, нужно вспомнить о том, как работает кэш:

1. В нём хранятся копии данных, к которым происходит наиболее частое обращение.
2. При обращении к данным процессор пытается найти их сначала в кэше.
3. Если данные в кэше не найдены, происходит обращение к ОЗУ по шине и переписывание данных из памяти в кэш. Далее они передаются процессору.

Обращение к ОЗУ занимает много времени. Считывание (из ОЗУ) и запись (в кэш) требуют обращение к шине, поэтому тоже работают долго.
Следовательно, необходимо свести количество этих действий к минимуму.

Быстрая работа функций на списке, находящемся в массиве, объясняется тем, что в кэш записывается строка, кратная степени двойки. Таким образом, для заполнения строки, кроме нужной области памяти, захватывается соседнее адресное пространство, в котором записаны какие-то служебные данные, которые не пригодятся нам в дальнейшем. Если же дозаполнить эту строку элементами, которые должны быть записаны в кэш позже, мы уменьшим количество обращений к ОЗУ (так как сначала поиск выполнятся в кэше, а затем в ОЗУ) и записи данных из ОЗУ в кэш.

Рассмотрим работу функций для 1-ого и 2-ого случаев на небольших списках, помещающихся в кэш (все функции выполняются для списков размера N, где (N=1000; N<=NMAX; N+=N/5), NMAX = 20000 (максимальная длина списков структур) , LEN = 1000 (размер выделяемого списка свободных элементов empty)), и получим следующие графики: 1, 2, 3.

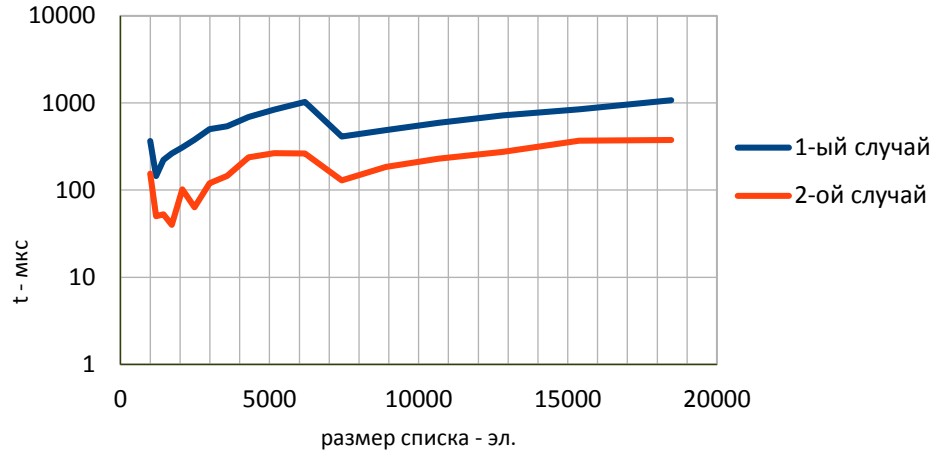


Иллюстрация 1: Работа функций malloc и pushafter в обоих случаях

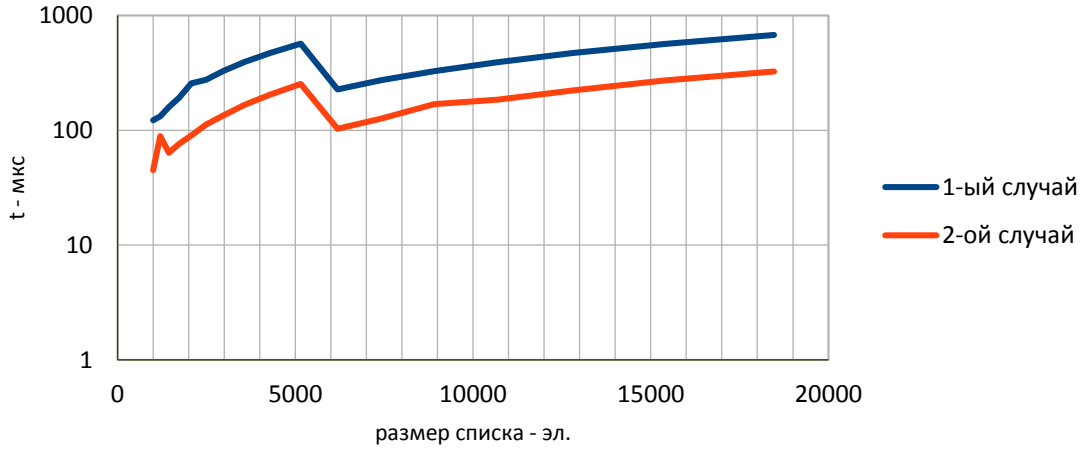
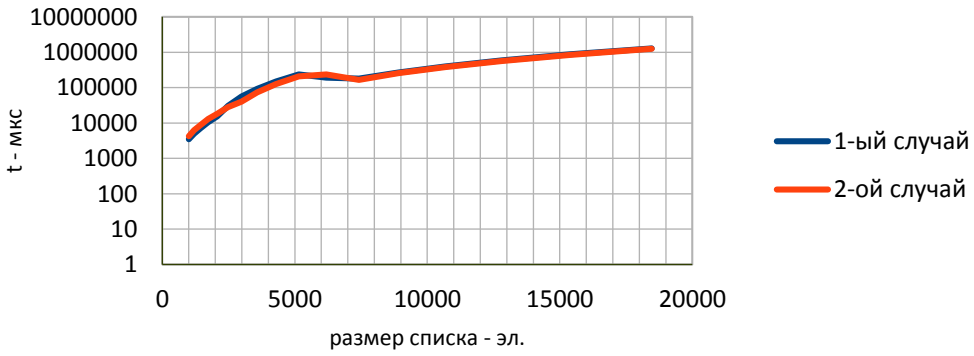


Иллюстрация 2: Работа функций delete в обоих случаях

Иллюстрация 3: Работа функций search в обоих случаях



Из графиков malloc и

delete видно, что на небольших размерах списка функции, написанные вручную, работают немного лучше, чем встроенные.

Скорость поиска search в обоих случаях практически одинаковая, потому что при маленьких массивах в кэш служебные данные почти не попадают.

В **третьем** случае **NEWMALLOC** и **DELETE** реализованы также, как и во втором, но список представляет собой массив структур:

```
struct Biglist {
   int elem[LEN];
   int prev[LEN];
   int next[LEN];
};                                                    //размер структуры = 12000 б
```

где каждый elem[i], prev[i], next[i] является соответственно полем elem, prev, next элемента i из списка, LEN = 1000, роль указателей играют индексы.

Реализация функции **NEWMALLOC** для данного варианта такая же как и в предыдущем (втором) случае: работа со списком пустых элементов «empty» аналогична, элементы из него возвращаются в return данной функции.

Иллюстрация работы функции NEWMALLOC для данного варианта:

```
int IS_EMPTY(struct Root *pr) {
   return pr->prev == -1 && pr->next == -1 ? 1 : 0;
}

int ENDEMPTY() {
   return IS_EMPTY( &empty );
}

int NEWMALLOC()
{
   if ( ENDEMPTY() ) {                                // если список свободных
                                                      // элементов пустой =>
                                                      // выделяем ему память
      int i, n = LEN*p_max;
      if ( p_max >= LENMAS ) {                        // если вышли за пределы
         printf ("Haven't free elements");            // заданной максимальной
         return -1;                                   // длины массива структур
      }
      struct Biglist *tmp = (struct Biglist*)malloc( sizeof(struct Biglist) );
      if ( !tmp ) {                                   // если не удалось выделить
         printf ("Problems with heap");               // память из кучи
         return -2;
      }
         for ( i = 0; i<LEN; i++ ) {
            tmp->elem[i] = 0;
            tmp->prev[i] = n + i - 1;
            tmp->next[i] = n + i + 1;
```

```
        }
        tmp->prev[0] = -1;
        tmp->next[LEN-1] = -1;
        p[p_max++] = tmp;                          // добавляем структуру в
        empty.next = n;                            // массив структур и
        empty.prev = LEN + n — 1;                  // закольцовываем empty
    }                                              // на данный список p

    int buf = empty.next;                          // иначе предоставляем
    empty.next = NEXT(buf, &empty);                // элемент из пустого (empty)
    PREV(NEXT(buf, &empty), &empty) = -1;
    return buf;
}
```

Реализация функции **DELETE** так же аналогична второму случаю: удаляем данный элемент из списка структур «p» и добавляем его в список свободных элементов как следующий элемент «empty».

Иллюстрация работы функции DELETE для данного варианта:

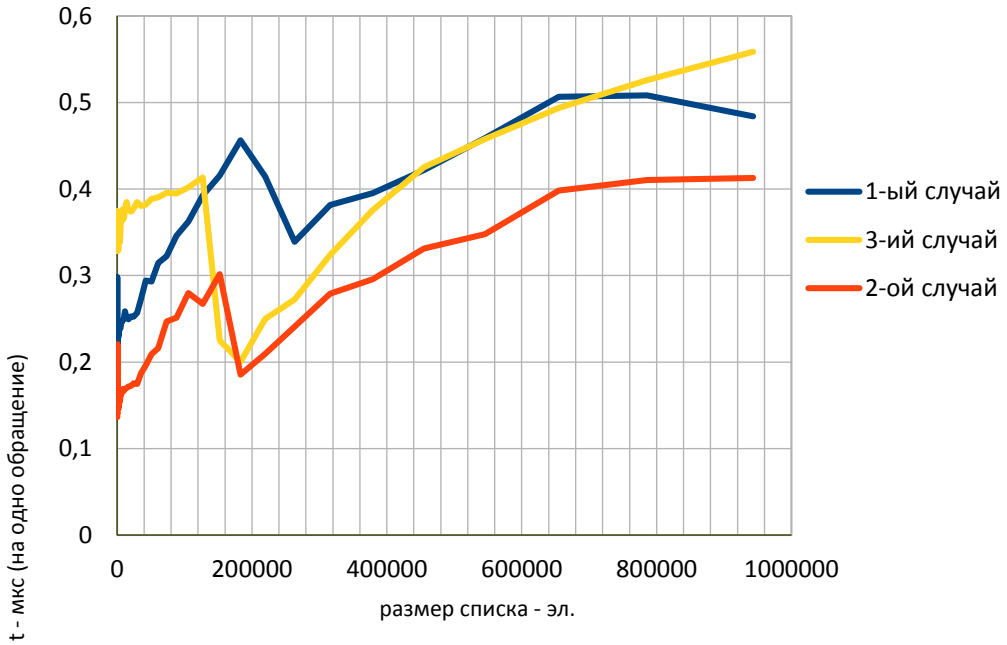
```
void DELETE( int ind ) {
    if ( ind == -1 ) return;
    struct Root dummy = { -1, -1 };
    int p = PREV( ind, &root ), n = NEXT( ind, &root );
    ELEM(ind) = 0;                                 // удаляем элемент из списка p
    PREV(n, &root) = p;
    NEXT(p, &root) = n;

    PREV(ind, &dummy) = -1;                        // добавляем его в список
    NEXT(ind, &dummy) = n = empty.next;            // свободных
    empty.next = ind;
    PREV(n, &empty) = ind;
    return;
}
                                                   // ELEM(), PREV(), NEXT() -
```

функции обращения к полям списка p : p[id/LEN]->elem[id%LEN], p[id/LEN]->prev[id%LEN], p[id/LEN]->next[id%LEN] соответственно.

Теперь **рассмотрим** работу функций для 1-ого, 2-ого и 3-его случаев на списках с максимальной длиной — 1000000, не влезающих в кэш, и 20000 (для поиска) и получим

графики 4, 5, 6. Следует отметить, что для функций malloc (+pushafter), delete по оси Y указано время, потраченное на один вызов функции.

Иллюстрация 4: Работа функций malloc и pushafter в трёх случаях, для списка максимальной длиной 1000000 элементов

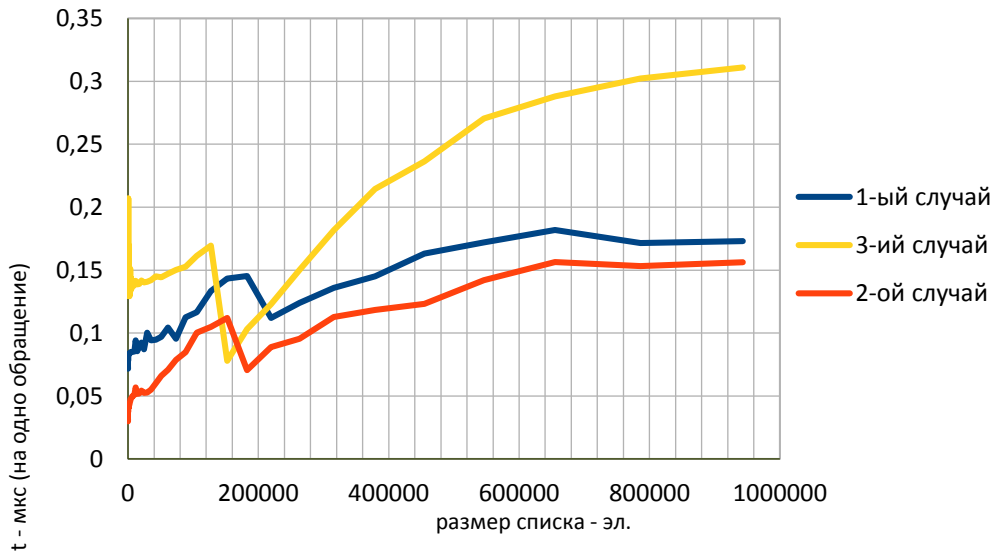


Видно, что malloc в первом случае работает хуже. Большая трата времени на ф-ию объясняется тем, что она выделяет память из кучи: элементы памяти располагаются далеко друг от друга, и кэш не будет эффективно с ними работать. Во втором же случае malloc выделяет список свободных элементов, закольцовывает его на основной массив и, таким образом, все элементы структуры находятся в одной области памяти. Функции malloc и pushafter третьего случая вначале показывают неплохой результат, но после 400000-ого элемента списка работают дольше, чем функии и 1-ого, и 2-ого случаев. Объясняется вычислительной сложностью обращения к полям списка elem, prev, next:

```
static inline int* P_NEXT( int id, struct Root *pr )
{
   return id == -1 ? &(pr->next) : &(p[id/LEN]->next[id%LEN]);
}
```

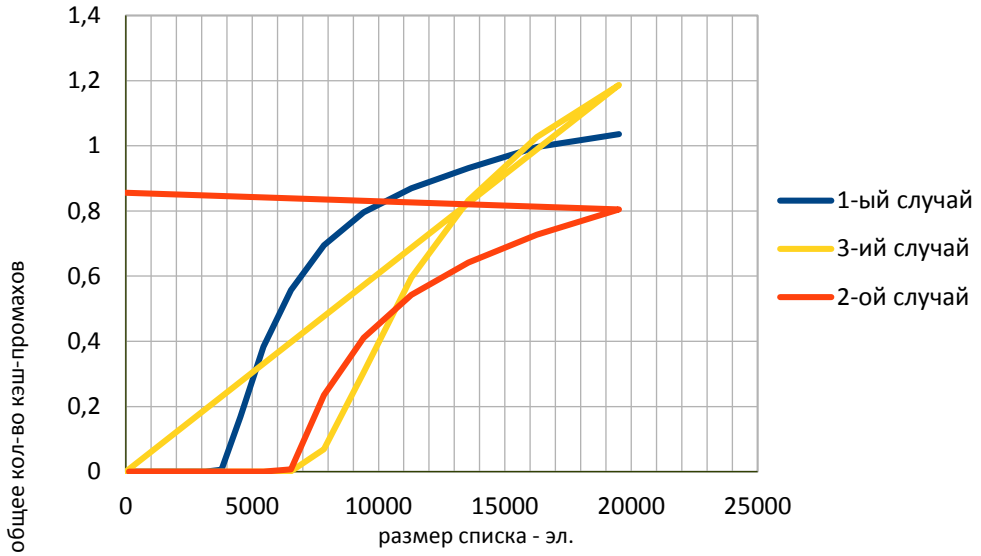
```
static inline int* P_PREV( int id, struct Root *pr )
{
   return id == -1 ? &(pr->prev) : &(p[id/LEN]->prev[id%LEN]);
}
static inline int* P_ELEM( int id )
{
   return id == -1 ? (int*)0 : &(p[id/LEN]->elem[id%LEN]);
}

#define ELEM(n) (*P_ELEM(n))
#define NEXT(n,pr) (*P_NEXT(n,pr))
#define PREV(n,pr) (*P_PREV(n,pr))
```

Иллюстрация 5: Работа функций delete в трёх случаях, для списка максимальной длиной 1000000 элементов



Функция delete во втором случае работает немного быстрее, так как добавляя элемент в список, находящийся в массиве, мы увеличиваем производительность. Долгая работа функции в 3-ем случае также объясняется тратой времени на вычисления.

Иллюстрация 6: Работа функций search в трёх случаях, для списка максимальной длиной 20000 элементов, размер кэша — 64Кб, ассоциативность 8 (считаем, что полностью ассоциативный, т. е. вытесняется самая старая строка), размер строки кэша - 64 б.



Функция поиска выполняется долго в обоих случаях, так как для нахождения элемента требуется обход всего списка. Однако 1-й вариант показал результат хуже по сравнению со 2-ым , потому что при больших размерах списков кэшируется много служебных данных, которые не пригодятся в дальнейшем.

Касательно графика работы функции search в третьем случае.

1. Стоит отметить, что после работы функций malloc и pushafter список загрузится в кэш, поэтому при первых вызовах search кэш-промахов не будет, что видно на графиках.
2. График функции в 3-ем случае начинает расти позже, чем во втором. Объясняется тем, что в отличие от 3-его, во 2-ом случае в кэш будут попадать структуры с полем prev, которое мы не используем в поиске: поле занимает третью часть структуры и график растёт быстрее.
3. Кэш-промахов в 3-ем случае получаем в 2 раза больше, чем во 2-ом, так как структура 3-его случая состоит из массивов полей. Соответственно, чтобы обратиться к двум полям elem и next, нужно считать 2 строки кэша (оба массива не могут находиться в одной строке).

Количество кэш-промахов при заполнении кэша можно вычислить формулой:

пусть K — размер кэша, L — размер строки кэша, N — размер списка, S — размер структуры списка, P — кол-во промахов . Тогда при N < K/S

$$P = S*N/L$$