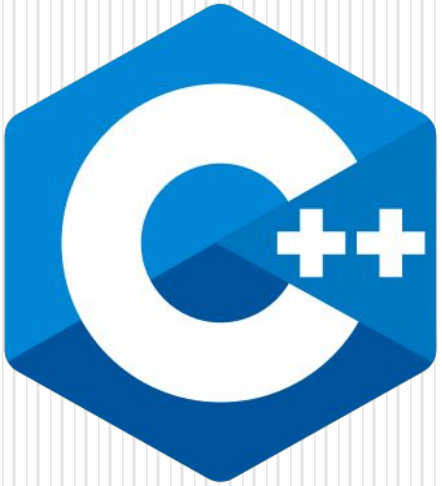
Лекция 9

Динамические структуры данных



Содержание

Динамические структуры данных

Любая программа предназначена для обработки данных, от способа организации которых зависят алгоритмы работы. Поэтому выбор структур данных должен предшествовать созданию алгоритмов.

Память под данные выделяется либо на этапе компиляции (в этом случае необходимый объем должен быть известен до начала выполнения программы, то есть задан в виде константы), либо во время выполнения программы с помощью операции new или функции malloc (необходимый объем должен быть известен до распределения памяти).

В обоих случаях выделяется непрерывный участок памяти. Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память выделяется по мере необходимости отдельными блоками, связанными друг с другом с помощью указателей. Такой способ организации данных называется динамическими структурами данных.

Динамические структуры данных

Из динамических структур в программах чаще всего используются **линейные списки**, **стеки**, **очереди** и **бинарные деревья**.

Они различаются способами связи отдельных элементов и допустимыми операциями. Динамическая структура может занимать несмежные участки оперативной памяти.

Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки, поскольку упорядочивание динамических структур не требует перестановки элементов, а сводится к изменению указателей на эти элементы.

Элемент любой динамической структуры данных представляет собой структуру (struct), содержащую по крайней мере два поля: для хранения данных и для указателя. Полей данных и указателей может быть несколько. Поля данных могут быть любого типа: **основного, составного или типа указатель.**

Линейные списки

Самый простой способ связать множество элементов - сделать так, чтобы каждый элемент содержал ссылку на следующий. Такой список называется однонаправленным (односвязным). Если добавить в каждый элемент вторую ссылку - на предыдущий элемент, получится двунаправленный, если последний элемент связать указателем с первым, получится кольцевой список.

Каждый элемент списка содержит ключ, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных.

Например, если создается линейный список из записей, содержащих фамилию, год рождения, стаж работы и пол, любая часть записи может выступать в качестве ключа

Операции над списками

Над списками можно выполнять следующие операции:

- начальное формирование списка (создание первого элемента);
- добавление элемента в конец списка;
- чтение элемента с заданным ключом;
- вставка элемента в заданное место списка (до или после элемента с заданным ключом);
- удаление элемента с заданным ключом;
- упорядочивание списка по ключу.

Линейные односвязные списки

Линейный список - это динамическая структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.

Из определения следует, что каждый элемент списка содержит поле данных (Data) (оно может иметь сложную структуру) и поле ссылки на следующий элемент (next). Поле ссылки последнего элемента должно содержать пустой указатель (NULL).

Так как ссылка всего одна (только на следующий элемент), то такой список является односвязным. Когда говорят о линейном списке, то, как правило, подразумевают именно односвязный список.

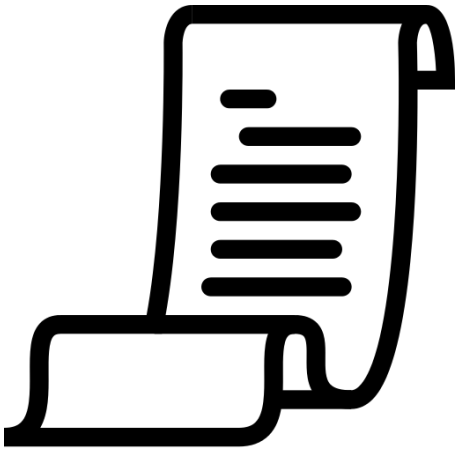
Например, необходимо сформировать список, содержащий целые числа 3, 5, 1, 9. Для решения этой задачи при работе с динамическими структурами данных можно рекомендовать следующий порядок действий.

Прежде всего, необходимо определить две структуры:

- структура, содержащая характеристики данных, то есть все те поля с данными, которые необходимы для решения поставленной задачи (в нашем случае имеется всего одно поле целого типа). Назовём эту структуру Data;
- структура, содержащая поле типа Data и поле - адрес последующего элемента next. Вторую структуру назовём List.

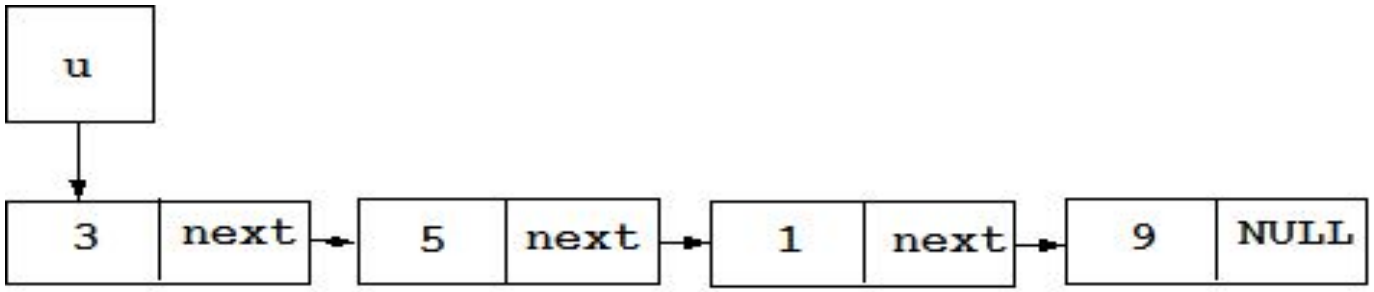
Тексты этих структур необходимо расположить в начале программы (до main() и других функций).

```
struct Data
{
  int a;
};
struct List
{
  Data d;
  List *next;
};
```



Такой подход позволит в дальнейшем изменять в широких пределах структуру с собственно данными, никак не затрагивая при этом основную структуру List.

Итак, мы описали структурный тип, с помощью которого можно создать односвязный список. Графически создаваемый список можно изобразить так, как это показано на рисунке ниже:



- В программе (обычно в функции main()) следует определить указатель на начало будущего списка:

List *u = NULL;

Пока список пуст, и указатель явно задан равным константе **NULL**.

- Выполняем первоначальное заполнение списка.

Создадим первый элемент:

```
u = new List;  // Выделяем память под элемент списка
u->d.a = 3;    // Заполняем поля с данными
               // (здесь это всего одно поле)
u->next = NULL;// Указатель на следующий элемент пуст
```

Продолжим формирование списка, добавляя новые элементы в его конец. Для удобства заведём вспомогательную переменную-указатель, которая будет хранить адрес последнего элемента списка:

```
List *x;
x = u; // Сейчас последний элемент списка совпадает с его началом
```

Таким образом, к области памяти можно обратиться через два указателя.

Выделяем место в памяти для следующего элемента списка и перенаправляем указатель x на выделенную область памяти:

x->next = new List;

x = x->next;

Затем определяем значение этого элемента списка:

x->d.a = 5;

x->next = NULL;

Этот процесс продолжаем до тех пор, пока не будет сформирован весь список.

Действия со сформированным списком. Сформировав начальный список, можно выполнять с ним различные действия. Рекомендуется каждую операцию со списком оформлять в виде отдельной функции. Такой подход заметно упрощает разработку программы и возможную её дальнейшую модификацию. Понятно, что и только что рассмотренное формирование начального списка также лучше записать в виде функции.

Просмотр списка

Просмотр списка - осуществляется последовательно, начиная с его начала. Указатель последовательно ссылается на первый, второй, и т.д. элементы списка до тех пор, пока весь список не будет пройден. Приведём пример реализации просмотра, например, для вывода списка на экран монитора:

```
void Print(List *u)
{
  List *p = u;
  cout << "Spisok:" << endl;
  while(p)
  {
    cout << p->d.a << endl;
    p = p->next;
  }
}
```

Обратиться к функции можно так:

```
Print(u);
```

Здесь и далее в примерах u - это указатель на начало списка.

Поиск первого вхождения в список элемента

```
List * Find(List *u, Data &x)
{
  List *p = u;
  while(p)
  {
    if(p->d.a == x.a) // условие для поиска
      return p;
    p = p->next;
  }
  return 0;
}
```

Возможный вызов функции:

```
List * v = Find(u, x);
```

где **x** - объект типа **Data**.

Вставка нового элемента

```
void Insert(List **u, Data &x)
{
  // вставка в список одного элемента перед элементом,
  // меньшим или равным данному x
  List *p = new List;
  p->d.a = x.a;
  if(*u == 0) // исходный список пуст - вставка в начало
  {
    p->next = 0;
    *u = p;
    return;
  }
  List *t = *u;
  if(t->d.a >= p->d.a) // исходный список не пуст -
                       // вставка в начало
  {
    p->next = t;
    *u = p;
    return;
  }
  List *tl = t->next;
  while(tl)
  {
    if(t->d.a < p->d.a && p->d.a <= tl->d.a)
      { // вставка в середину
      t->next = p;
      p->next = tl;
      return;
    }
    t = tl;
    tl = tl->next;
  }
  t->next = p; // добавляем в конец списка
  p->next = 0;
}
```

Возможный вызов функции:

Insert(&u, x)

где **x** - объект типа **Data**.

Эта функция позволяет сразу формировать упорядоченный список.

Удаление элемента из линейного списка

```
void Delete(List **u, Data &x)
{
  if(*u == 0) // исходный список пуст - удалять нечего!
  {
    return;
  }
  List *t = *u;
  if(t->d.a == x.a) // исходный список не пуст -
                    // удаляется начало
  {
    *u = t->next;
    delete t;
    return;
  }
  List *tl = t->next;
  while(tl)
  {
    if(tl->d.a == x.a)
    // исходный список не пуст -
    //удаляется не первый элемент
    {
      t->next = tl->next;
      delete tl;
      return;
    }
    t = tl;
    tl = tl->next;
  }
}
```

Возможный вызов функции:

Delete(&u, x);

где **x** - объект типа **Data**.

Удаление (очистка) всего списка

Когда данные, хранящиеся в списке, становятся ненужными, можно очистить весь список, т.е. освободить память, которую занимали все элементы списка.

Выполнять эту операцию желательно сразу после того, как список стал не нужен. Реализация этой операции может быть такой:

```
void Clear(List **u)
{
  // удаление (очистка) всего списка
  if(*u == 0) return;
  List *p = *u;
  List *t;
  while(p)
  {
    t = p;
    p = p->next;
    delete t;
  }
  *u = 0;
}
```

Возможный вызов функции:

```
Clear(&u);
```


Стеки

Стек - это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняются с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (last in - first out, последним пришел - первым ушел). Стек проще всего представить как закрытую с одного конца узкую трубу, в которую бросают мячи.

Достать первый брошенный мяч можно только после того, как вынуты все остальные. Сегмент стека назван так именно потому, что память под локальные переменные выделяется по принципу LIFO. Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах.

Функция помещения в стек по традиции называется push, а выборки - pop. Указатель для работы со стеком (top) всегда ссылается на его вершину.
Пример программы, которая формирует стек из пяти целых чисел (1, 2, 3, 4, 5) и выводит его на кран:

```
#include "pch.h"
#include <iostream>
struct Node{
int d;
Node *p;
};
Node * first(int d);
void push(Node **top, int d);
int pop(Node **top);
//--------------------------------------------
int main(){
Node *top = first(1);
for (int i = 2; i<6; i++)push(&top, i);
while (top)
cout << pop(&top) << ' ';
return 0;
}
```

```
// Начальное формирование стека
Node * first(int d){
Node *pv= new Node;
pv->d = d;
pv->p = 0;
return pv;
}
// Занесение в стек
void push(Node **top, int d){
Node *pv= new Node;
pv->d = d;
pv->p = *top;
*top = pv;
}
// Выборка из стека
int pop(Node **top){
int temp = (*top)->d;
Node *pv= *top;
*top = (*top)->p;
delete pv;
return temp;
}
```

Очереди

Очередь - это частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка - из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди.

Очередь реализует принцип обслуживания FIFO (first in - first out, первым пришел - первым ушел). Очередь проще всего представить себе, постояв в ней час-другой. В программировании очереди применяются, например, в моделировании, диспетчеризации задач операционной системой, буферизованном вводе/выводе.

Функция помещения в конец очереди называется add, а выборки - del. Указатель на начало очереди называется pbeg, указатель на конец - pend.

Пример программы, которая формирует очередь из пяти целых чисел и выводит его на кран

```
#include "pch.h"
#include <iostream>
struct Node{
int d;
Node *p;
};
Node * first(int d);
void add(Node **pend, int d);
int del(Node **pbeg);
//-----------------------------------------
int main(){
Node *pbeg = first(1);
Node *pend = pbeg;
for (int i = 2; i<6; i++)add(&pend, i);
while (pbeg)
cout << del(&pbeg) << ' ';
return 0;
}
```

```
// Начальное формирование очереди
Node * first(int d){
Node *pv= new Node;
pv->d = d;
pv->p = 0;
return pv;
}
//-----------------------------------------
----
// Добавление в конец
void add(Node **pend, int d){
Node *pv= new Node;
pv->d = d;
pv->p = 0;
(*pend)->p = pv;
*pend = pv;}
// Выборка
int del(Node **pbeg){
int temp = (*pbeg)->d;
Node *pv= *pbeg;
*pbeg = (*pbeg)->p;
delete pv;
return temp; }
```

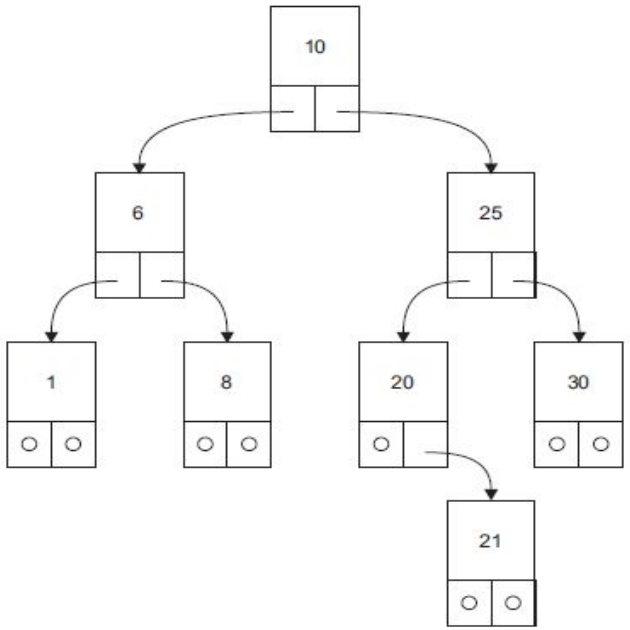

Бинарные деревья

Бинарное дерево - это динамическая структура данных, состоящая из узлов, каждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется корнем дерева. Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие - потомками.

Высота дерева определяется количеством уровней, на которых располагаются его узлы. Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева - больше, оно называется деревом поиска. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле.

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью рекурсивных алгоритмов. Например, функцию обхода всех узлов дерева в общем виде можно описать так:

function way_around (дерево){
way_around (левое поддерево)
посещение корня
way_around (правое поддерево)



Бинарное дерево

Таким образом, деревья поиска можно применять для сортировки значений. При обходе дерева узлы не удаляются.

Для бинарных деревьев определены операции:

- включения узла в дерево;
- поиска по дереву;
- обхода дерева;
- удаления узла.

Для каждого рекурсивного алгоритма можно создать его не рекурсивный эквивалент.

В приведенной ниже программе реализована не рекурсивная функция поиска по дереву и рекурсивная функция обхода дерева. Первая функция осуществляет поиск элемента с заданным ключом. Если элемент найден, она возвращает указатель на него, а если нет - включает элемент в соответствующее место дерева и возвращает указатель на него. Для включения элемента необходимо помнить пройденный по дереву путь на один шаг назад и знать, выполняется ли включение нового элемента в левое или правое поддерево его предка.

Программа формирует дерево из массива целых чисел и выводит его на экран

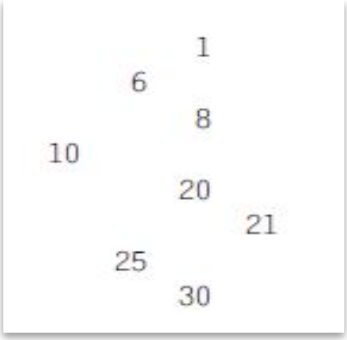
```
#include "pch.h"
#include <iostream>
struct Node{
int d;
Node *left;
Node *right;
};
Node * first(int d);
Node * search_insert(Node *root, int d);
void print_tree(Node *root, int l);
//--------------------------------------
int main(){
int b[] = {10, 25, 20, 6, 21, 8, 1, 30};
Node *root = first(b[0]);
for (int i = 1; i<8; i++)search_insert(root, b[i]);
print_tree(root, 0);
return 0;
}
```

```
// Формирование первого элемента дерева
Node * first(int d){
Node *pv= new Node;
pv->d = d;
pv->left = 0;
pv->right = 0;
return pv;
}
//--------------------------------------
// Поиск с включением
Node * search_insert(Node *root, int d){
Node *pv= root, *prev;
bool found = false;
while (pv&& !found){
prev= pv;
if (d == pv->d) found = true;
else if (d < pv->d) pv = pv->left;
else pv= pv->right;
}
```

Программа формирует дерево из массива целых чисел и выводит его на экран

- if (found) return pv;
- // Создание нового узла:
- Node *pnew = new Node;
- pnew->d = d;
- pnew->left = 0;
- pnew->right = 0;
- if (d < prev->d)
- prev->left = pnew; // Присоединение к левому поддереву предка
- else
- prev->right = pnew; // Присоединение к правому поддереву предка
- return pnew;
- }

- // Обход дерева
- void print_tree(Node *p, int level){
- if (p){
- print_tree(p->left, level + 1); // вывод левого поддерева
- for (int i = 0; i<level; i++)cout << " ";
- cout << p->d << endl; // вывод корня поддерева
- print_tree(p->right, level + 1); // вывод правого поддерева
- }



Результат работы программы

Реализация динамических структур с помощью массивов

Если максимальный размер данных можно определить до начала использования и в процессе работы он не изменяется (например, при сортировке содержимого файла), более эффективным может оказаться однократное выделение непрерывной области памяти. Связи элементов при этом реализуются не через указатели, а через вспомогательные переменные или массивы, в которых хранятся номера элементов.

Проще всего реализовать таким образом стек. Кроме массива элементов, соответствующих типу данных стека, достаточно иметь одну переменную целого типа для хранения индекса элемента массива, являющегося вершиной стека. При помещении в стек индекс увеличивается на единицу, а при выборке - уменьшается. Для реализации очереди требуются две переменных целого типа - для хранения индекса элементов массива, являющихся началом и концом очереди.

Для реализации линейного списка требуется вспомогательный массив целых чисел и еще одна переменная, например:

- 10 25 20 6 21 8 1 30 - массив данных;
- 1 2 3 4 5 6 7 -1 - вспомогательный массив;
- 0 - индекс первого элемента в списке.

i-й элемент вспомогательного массива содержит для каждого i-го элемента массива данных индекс следующего за ним элемента. Отрицательное число используется как признак конца списка. Тот же массив после сортировки:

- 10 25 20 6 21 8 1 30 - массив данных;
- 2 7 4 5 1 0 3 -1 - вспомогательный массив;
- 6 - индекс первого элемента в списке.

Для создания бинарного дерева можно использовать два вспомогательных массива (индексы вершин его правого и левого поддерева). Отрицательное число используется как признак пустой ссылки. Например, дерево на рисунке 5.3 можно представить следующим образом:

- 10 25 20 6 21 8 1 30 - массив данных;
- 3 2 -1 6 -1 -1 -1 -1 - левая ссылка;
- 1 7 4 5 -1 -1 -1 -1 - правая ссылка.

Память под такие структуры можно выделить либо на этапе компиляции, если размер можно задать константой, либо во время выполнения программы, например:

```
struct Node{
Data d; // тип данных Data должен быть определен ранее
int i;
};
Node spisok1[1000]; // на этапе компиляции
Node *pspisok2 = new Node[m]; // на этапе выполнения
```


Контрольные вопросы

1. Какие операции можно выполнять над списками?
2. Что такое стек?
3. Что такое очередь?
4. Как определяется высота дерева?
5. Какие операции определены для бинарных деревьев?

Список литературы

- Павловская Т.А. С/С++. Программирование на языке высокого уровня / Т. А. Павловская. - СПб.: Питер, 2004. - 461 с.: ил.
- Павловская Т.А. С/С ++. Структурное программирование: Практикум / Т.А. Павловская, Ю.А. Щупак. СПб.: Питер, 2007. - 239 с.: ил.
- Павловская Т. А., Щупак Ю. А. С++. Объектно-ориентированное программирование: Практикум. - СПб.: Питер, 2006. - 265 с: ил.
- Кольцов Д.М. 100 примеров на Си. - СПб.: “Наука и техника”, 2017 - 256 с.
- 5 Доусон М. Изучаем С++ через программирование игр. - СПб.: “Питер”, 2016. - 352.
- Седжвик Р. Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ. Роберт Седжвик. - К.: Издательство “Диасофт”, 2001. - 688с.
- Сиддкхартха Р. Освой самостоятельно С++ за 21 день. - М.: SAMS, 2013. - 651 с.
- Стивен, П. Язык программирования С++. Лекции и упражнения, 6-е изд. Пер. с англ. - М.: ООО "И.Д. Вильямс", 2012. - 1248 с.
- Черносвитов, А. Visual C++: руководство по практическому изучению / А. Черносвитов . - СПб. : Питер, 2002. - 528 с. : ил.

Список литературы

- Страуструп Б. Дизайн и эволюция языка C++. - М.: ДМК, 2000. - 448 с.
- Мейерс С. Эффективное использование C++. - М.: ДМК, 2000. - 240 с.
- Бадд Т. Объектно-ориентированное программирование в действии. - СПб: Питер, 1997. - 464 с.
- Лаптев В.В. С ++. Объектно-ориентированное программирование: Учебное пособие.- СПб.: Питер, 2008. - 464 с.: ил.
- Страуструп Б. Язык программирования C++. Режим доступа: http://8361.ru/6sem/books/Straustrup-Yazyk_programmirovaniya_c.pdf.
- Керниган Б., Ритчи Д. Язык программирования Си. Режим доступа: http://cpp.com.ru/kr_cbook/index.html.
- Герберт Шилдт: C++ базовый курс. Режим доступа: https://www.bsuir.by/m/12_100229_1_98220.pdf,
- Богуславский А.А., Соколов С.М. Основы программирования на языке Си++. Режим доступа: http://www.ict.edu.ru/ft/004246/cpp_p1.pdf.
- Линский, Е. Основы C++. Режим доступа: https://www.lektorium.tv/lecture/13373.
- Конова Е. А., Поллак Г. А. Алгоритмы и программы. Язык C++: Учебное пособие. Режим доступа: https://vk.com/doc7608079_489807856?hash=e279524206b2efd567&dl=f85cf2703018eeaa2