Задачи за едносвързан списък

Рая Георгиева

18 ноември 2020 г.

1 Основни понятия

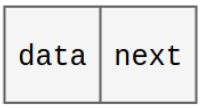
Свързаният списък е хомогена линейна стуктура. Линейността се определя не от физическото разположение на елементите в паметта (както при масивите), а от наличието на указатели във всеки един възел към следващия и/или предхождащия го възли. Характерни за свързаните списъци са ефективността при вмъкване на елемент и по-голямата сложност на достъп до елемент в сравнение с обикновените масиви.
По друг начин казано, свързаният списък е колекция от възли, които посредством връзките между тях имитират подреденост. Ще разглеждаме **едносвързан списък**. Тук, всеки възел притежава поле за данни и указател към следващия го възел.

Сложност на операциите:

- Достъп: O(n)
- Търсене: O(n)
- Вмъкване: O(1)
- Изтриване: O(1)

2 Реализация

Ще направим реализацията само на възела, като във функциите възелът ще си го мислим за цял лист. Много подобна е реализацията, ако имаме клас за листа с член данна възел, представляваща първия му елемент.



Фигура 1: Възел на едносвързан списък. **data** съдържа стойността на възела, тази променлива е от шаблонен тип. **next** съдържа адрес на следващата клетка и от тип указател към клетка.

```
template <typename T>
struct Node {
   T data;
   Node * next {nullptr};
};
```

Уточнения:

- При създаване на нов възел насочваме неговия *next* указател към *nullptr*, защото този възел трябва да представлява сам по себе си списък (от един елемент).
- Приемаме, че свързан списък е празен тогава, когато указателят сочещ към него е *nullptr*.

3 Задачи - списък

1. Да се напише функция pushFront, която добавя нов елемент в началото на списъка.

```
template <typename T>
void pushFront(Node<T> ** head, T elem)
{
   // your code here
}
```

2. Да се напише функция, която връща дължината на свързан списък.

```
template <typename T>
```

```
size_t length(Node<T> *head)
{
   // your code here
}
```

3. Да се напише функция, която връща броя на елементите на списък, равни на дадена стойност.

```
template <typename T>
size_t count(Node<T> *head, T const& searchFor)
{
   // your code here
}
```

4. Да се напише функция, която приема списък и число - индекс и връща елемента на позиция този индекс от списъка. Индексите започват от 0. Индексът е по-голям от броя на елементите, то нека функцията връща елемент от конкретния тип, създаден с конструктора по подразбиране.

5. Да се напише функция, която приема указаел към списък и изтрива елементите му.

```
template <typename T>
void deleteList(Node<T> *& head) // not ok, why?
{
   // your code ...
}
```

6. Да се напише фунция обратна на push, тоест функция pop, която премахва елемент от началото на списък.

```
template <typename T>
T pop(Node<T> **head)
```

```
{
   // your code ...
}
```

7. Да се напише функция, която приема индекс и стойност и вмъква възел с тази стойност в списъка на зададената позиция. Индексите приемаме, че започват от 0. Ако индексът е по-голям от броя на елементите в списъка, то добавяме накрая.

```
template <typename T>
void insertNth(Node<T> ** head, size_t index, T elem)
{
   // your code ...
}
```

8. Да се напише функция, която приема **възел** и го вмъква на подходящата позиция в сортиран списък, тоест на такава позиция, че данните на предходния възел са по-малки или равни, а данните на следващия възел са по-големи.

```
template <typename T>
void sortedInsert(Node<T> ** head, Node<T> * node)
{
   // your code ...
}
```

9. Да се напише функция, която реализира сортиране чрез вмъкване за едносвързан списък. (Hint: да се използва *sortedInsert*).

```
template <typename T>
void insertionSort(Node<T> ** head)
{
   // your code ...
}
```

10. Да се напише функция, която разделя свирзан списпък на два други списъка през средата.

```
template <typename T>
void frontBackSplit(Node<T> ** head, Node<T> ** front,
Node<T> ** back)
{
   // your code ...
}
```

11. Да се напише функция, която приема сортиран списък и премахва елементите, които се повтарят.

```
template <typename T>
void removeDuplicates(Node<T> ** head)
{
// your code ...
}
```

12. Да се напише функция, която приема два списъка и премества първия елемент на втория списък в началото на първия и извършва цялата указателна аритметика около това.

```
template <typename T>
void moveNode(Node<T> **dest, Node<T> **source)
{
   // your code ...
}
```

13. Да се напише функция, която разделя списък на други два списъка, като последователно слага в единия и в другия, тоест всички елементи на четни позиции са в единия списък, а всички с нечетни позиции са в другия.

```
template <typename T>
void alternatingSplit(Node<T> **head, Node<T> **aList,
Node<T> **bList)
{
   // your code ...
}
```

14. Да се напише функция, която приема два свързани списъка и ги слива в един списък, редувайки вземането от единия и от другия. Пробвайте да я напишете рекурсивно.

```
template <typename T>
Node<T> *shuffleMerge(Node<T> *a, Node<T> *b)
{
   // your code ...
}
```

15. Да се напише фунцкия, която приема два сортирани списъка и ги слива в един сортиран списък.

```
template <typename T>
void sortedMerge(Node<T> **merged, Node<T> *&a, Node<T>
 *&b)
{
   // your code ...
}
```

16. Напишете функция, която приема свързан списък го сортира по метода на сливането (merge sort).

```
template <typename T>
void mergeSort(Node<T> **head)
{
   // your code ...
```

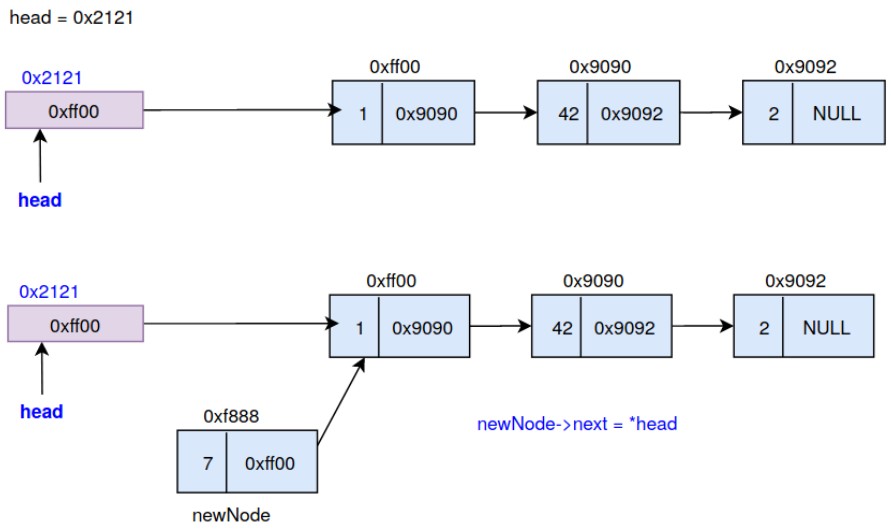
```
}
```

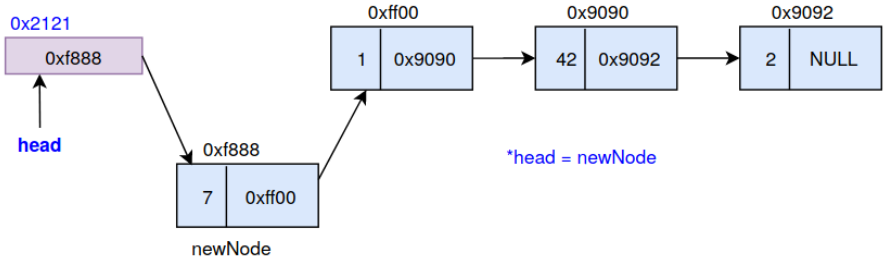
17. Да се напише функция, която приема свързан списък и обръща реда на елементите му. Опитайте се да я напишете рекурсивно.

4 Задачи - решения

1. Има една важна особенест при решението на тази задача, а именно, че след изпълнението head трябва да сочи към новото начало. Това е причината, поради която го подаваме като двоен указател - да можем да сменим адреса в него.

```
template <typename T>
void pushFront(Node<T> ** head, T elem)
{
   auto newNode = new Node<T>(elem);
   newNode->next = *head;
   *head = newNode;
}
```





2. За да решим тази задача трябва да си зададем няколко въпроса: 1. Как да се придвижваме напред по елементите на списъка? 2. Как да разберем кога листът е свършил? 3. Как да се справим с празните списъци?
   Отговорът на първия въпрос се съдържа в начина, по който се представят възлите: във всеки един има указател към следващия възел, тоест този *next* указател ние ще използваме за придвижване. За да постигнем това си заделяме променлива, в която ще пазим адрес на текущия възел, който разглеждаме. Нека я кръстим *current*. Първоначално тя е равна на head, тоест на адреса на първия елемент от списъка, или на *nullptr*.

```
Node<T> *current = head;
```

   Когато искаме да "преминем напред" просто променяме променливата *current* да сочи към следващия възел:

```
current = current->next
```

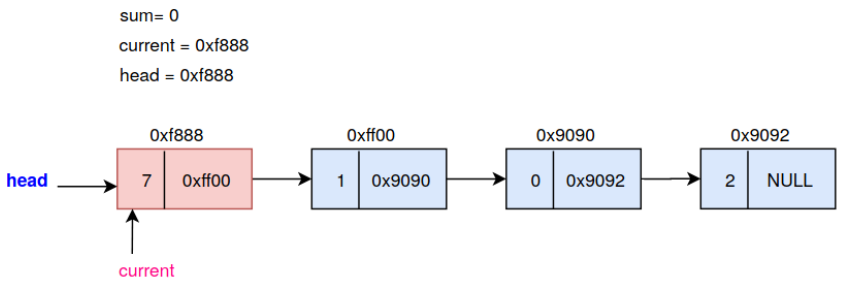
   Отговорът на втория въпрос вече е очевиден: последният елемент сме стигнали, когато сме стигнали елемент, чийто *next* указател сочи към *nullptr*, съответно проверката ни за край на обхождането е когато *current* има стойност *nullptr*. Във връзка с третия въпрос няма нужда да се прави нищо допълнително, ако самият указател *head* е *nullptr*, то ще изпаднем веднага в случая за последен елемент.
   Горната схема се повтаря често за най-различни задачи, като се променят само същинската част от сметките. Тук искаме да сметнем броя на елементите, тоест инициализираме дължината със стойност дължината на празния списък, тоест 0, и за всеки елемент увеличаваме тази дължина с 1.

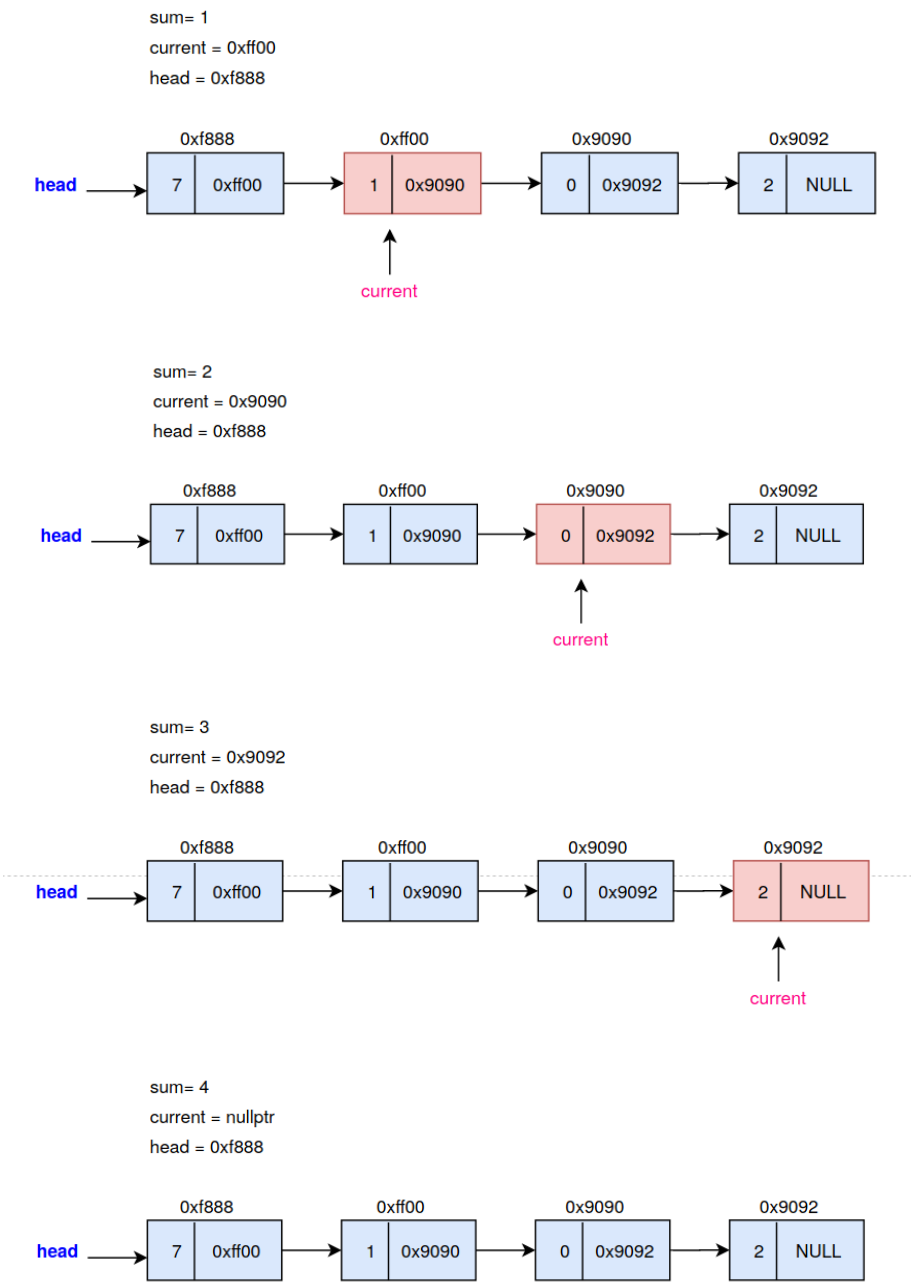
```
template <typename T>
size_t length(Node<T> *head)
{
   Node<T> *current = head;
   size_t length = 0;

   while (current != nullptr)
   {
      current = current->next;
      length++;
   }

   return length;
}
```

Алтернативно обхождането на свързан списък може да стане и с използването на for-цикъл по следния начин:

```
for (current = head; current != nullptr; current = current
    ->next)
{
// ...
}
```

Примерно действие на алгоритъма:





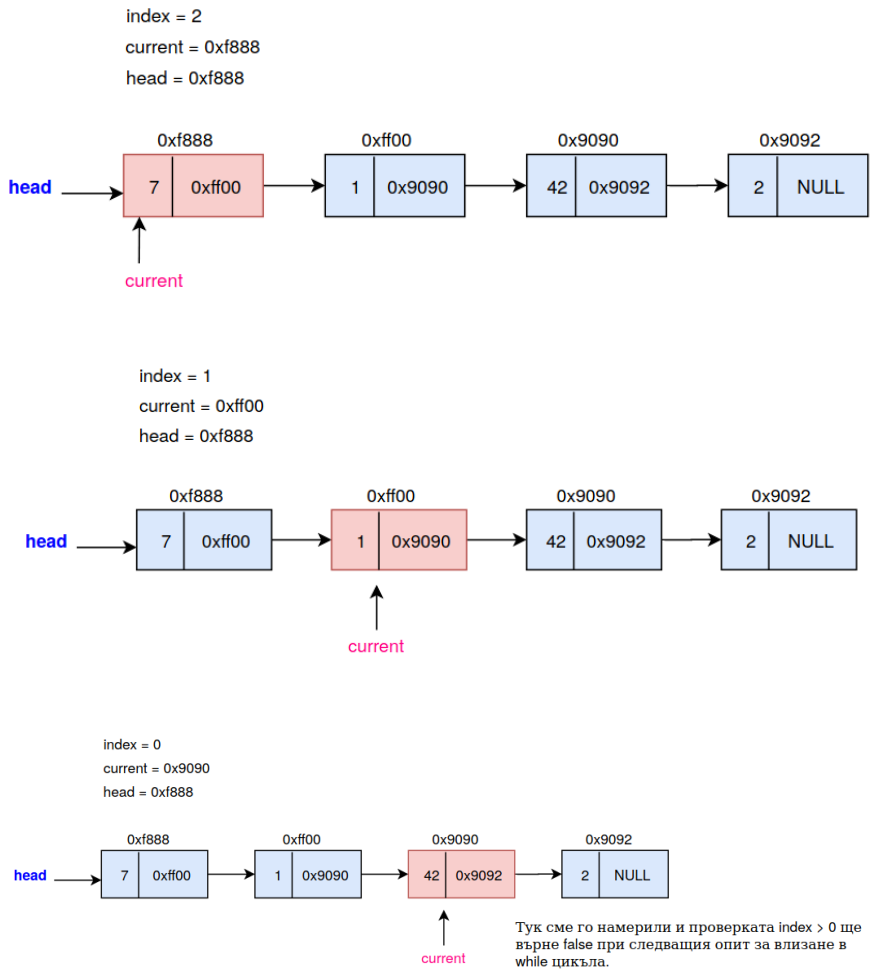
3. За тази задача идеята е същата като горната, но вместо на всяка стъпка да увеличаваме брояча, го увеличаваме само когато текущият възел има данни равни на търсеното.

```
template <typename T>
size_t count(const Node<T> * head, T searchFor)
{
   Node<T> *current = head;
   size_t count = 0;
   while (current)
   {
      if (current->data == searchFor)
      {
         count++;
      }
      current = current->next;
   }
   return count;
}
```

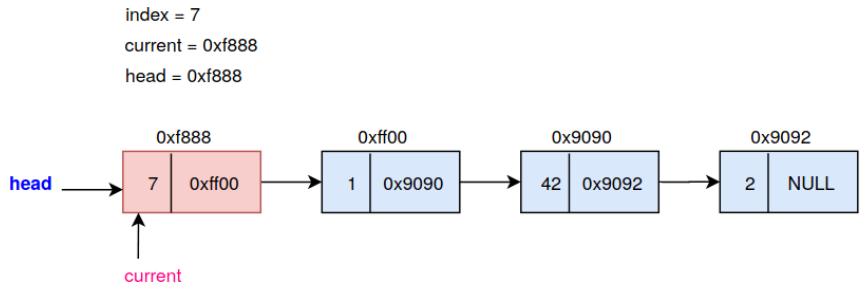
4. Обикаляме списъка докато: не намерим търсения индекс или не свърши списъка. За да намерим индекса, то ще намаляме подадения ни параметър докато не стане 0, местейки с една позиция напред от главата. Вижда се, че ако търсим индекс 0, то няма да има местене и си връщаме направо данните от главата. Накрая, ако index е 0, то значи сме го намерили и връщаме текущия елемент, ако пък е по-голям от 0, то значи подаденият ни индекс е бил по-голям от броя на елементите.

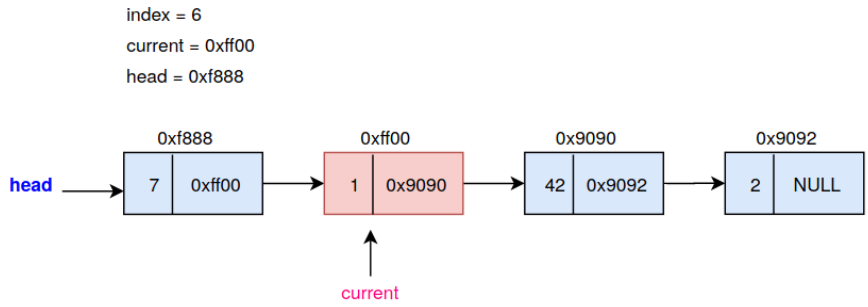
```
template <typename T>
T getNth(Node<T> *head, size_t index)
{
   Node<T> *current = head;
   while (current != nullptr && index > 0)
   {
      current = current->next;
      index--;
   }

   return index == 0 ? current->data : T();
}
```

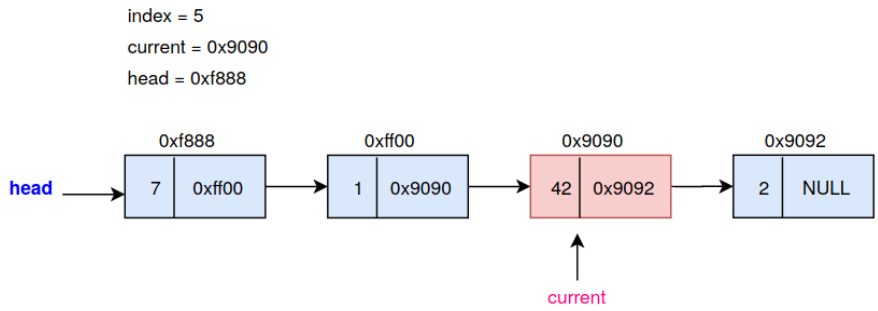
Пример, когато индексът е в граници:

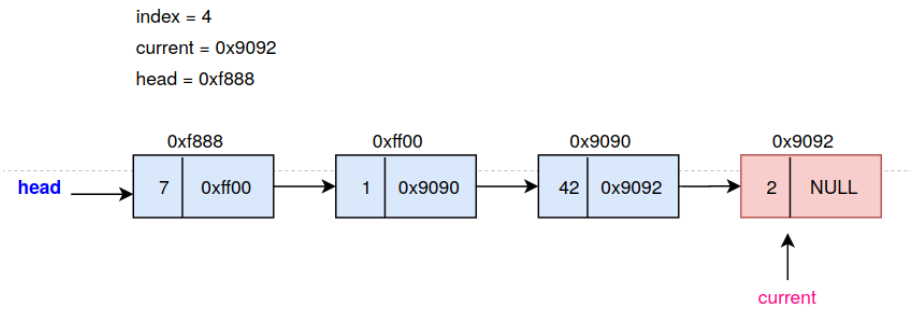


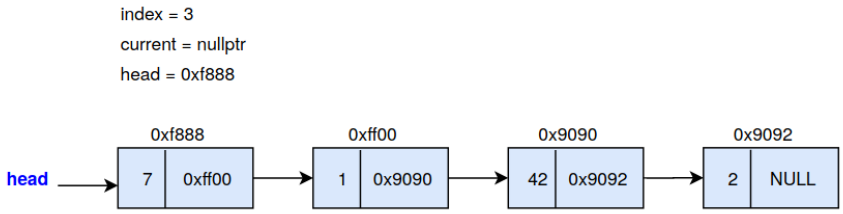
Пример, когато индексът е по-голям от броя на елементите на списъка:


index = 7
current = 0xf888
head = 0xf888
0xf888
0xff00
0x9090
0x9092
head
7
0xff00
1
0x9090
42
0x9092
2
NULL
current


index = 6
current = 0xff00
head = 0xf888
0xf888
0xff00
0x9090
0x9092
head
7
0xff00
1
0x9090
42
0x9092
2
NULL
current


index = 5
current = 0x9090
head = 0xf888
0xf888
0xff00
0x9090
0x9092
head
7
0xff00
1
0x9090
42
0x9092
2
NULL
current


index = 4
current = 0x9092
head = 0xf888
0xf888
0xff00
0x9090
0x9092
head
7
0xff00
1
0x9090
42
0x9092
2
NULL
current



Забележете, че списъкът свърши и *current* е *nullptr* проверката за *index == 0* е лъжа, тоест ще върнем *T()*.

5. Тук вместо указател използваме референция. Но идеята е същата, преминаваме през всеки елемент и го изтриваме. Ако изтрием елемента просто така:

```
delete current;
```

то ще имаме проблем, защото ще изгубим указателя към следващия елемент и няма как да го достъпим, затова се налага да запазим единия във временна променлива. Аз избрах да запазя текущия. Така пазя текущия, измествам напред *current* и после мога да изтрия запазеното.

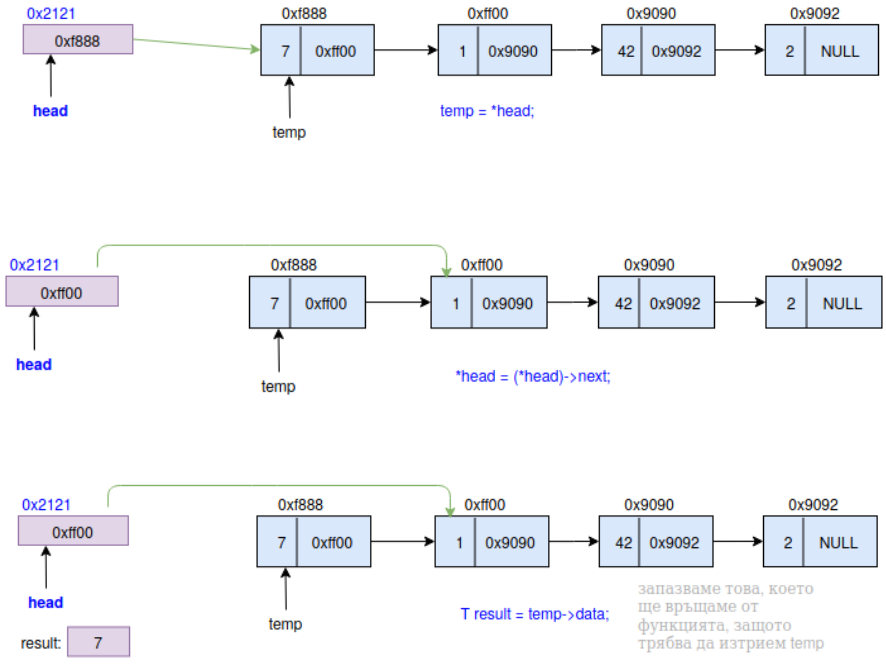
```
template <typename T>
void deleteList(Node<T> *& head) // not ok, why?
{
   Node<T> * current = head;
   while (current != nullptr)
   {
      Node<T> * temp = current;
      current = current->next;
      delete temp;
   }
   head = nullptr;
}
```

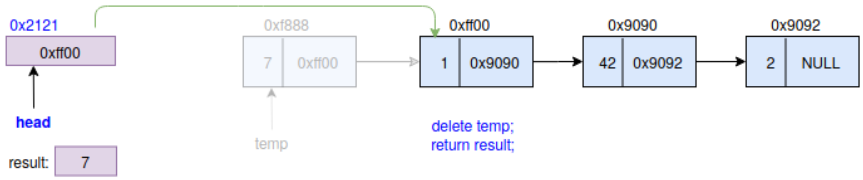
6. Тук трябва да се направи следното: да се изтрие първия възел от листа, да се промени head указателя. Тук трябва да се внимава за

това, че се изисква след изтриването стойността да се върне. По тази причина се налага да я запазим в една променлива предварително.

```
T pop(Node<T> **head)
{
   if (*head == nullptr)
   {
      // empty list, return default constructor
      return T();
   }
   auto temp = *head;
   T result = temp->data;
   *head = (*head)->next;
   delete temp;
   return result;
}
```

Пример:





7. Ако подаденият ни индекс е по-голям от броя на елементите на списъка, то доабвяме накрая. По тази причина обикаляме списъка докато не свърши или докато не намерим индекса, който търсим. Тъй като ще вмъкваме е редно в една променлива да пазим елемента, след който ще вмъкваме (*prev*).
   За намирането на индекса правим променлива *idx*, която на всяко изместване увеличаваме с 1, докато не стигне желаната стойност (може и да намаляме подадената ни докато е по-голяма от 0). Затова условието при итериране е:

```
current != nullptr && idx < index
```

   Ако едното от двете се провали, спираме итерирането. Съответно след това трябва да направим проверка кое точно се е провалило, за да разберем защо е спряло. Друго нещо, за което трябва да внимаваме е добавянето на индекс 0 или добавянето в празен списък, защото при тези следва промяна на стойността, сочена от *head*. Можем лесно да проверим дали сме в тези случаи, като направим проверка на prev, ако той има стойност nullptr, то със сигурност вмъкваме в началото.

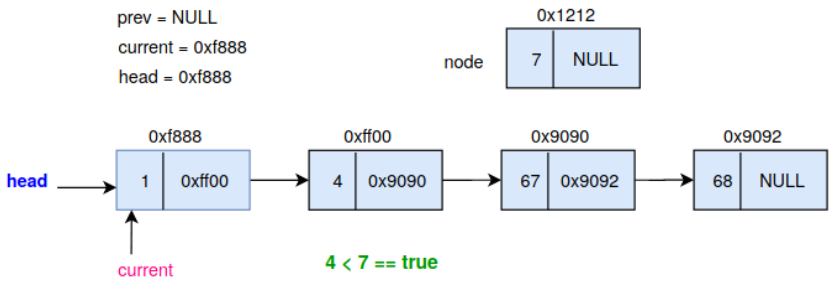
```
template <typename T>
void insertNth(Node<T> ** head, size_t index, T elem)
{
   Node<T> *newNode = new Node<T>(elem);
   size_t idx = 0;
   Node<T> *current = *head, *prev = nullptr;
   while (current != nullptr && idx < index)
   {
      prev = current;
      current = current->next;
      ++idx;
```

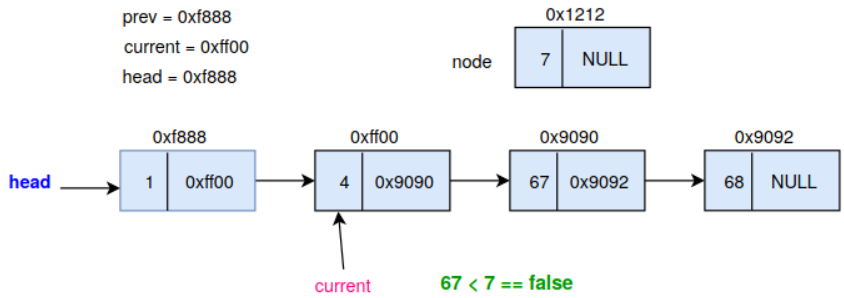
```
    }

    if (prev == nullptr)
    {
        // this means that we are inserting on index 0
        if (*head == nullptr)
        {
            // empty list
            *head = newNode;
        }
        else
        {
            // insert on front
            newNode->next = *head;
            *head = newNode;
        }
    }
    else
    {
        newNode->next = prev->next;
        prev->next = newNode;
    }
}
```

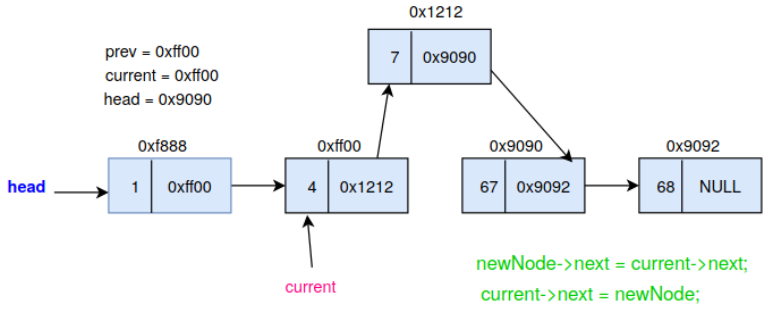
8. Тук имаме следните специални случаи: добавяне в началото, тоест възелът, който доабавяме е по-малък от първия възел и добавяне в празен списък. След това процедираме по-следния начин: намираме елемента, след който ще вмъкваме и след това извършваме самото вмъкване.

```
template <typename T>
void sortedInsert(Node<T> ** head, Node<T> * newNode)
{
    if (*head == nullptr || (*head)->data >= newNode->data)
    {
        newNode->next = *head;
        *head = newNode;
    }
```

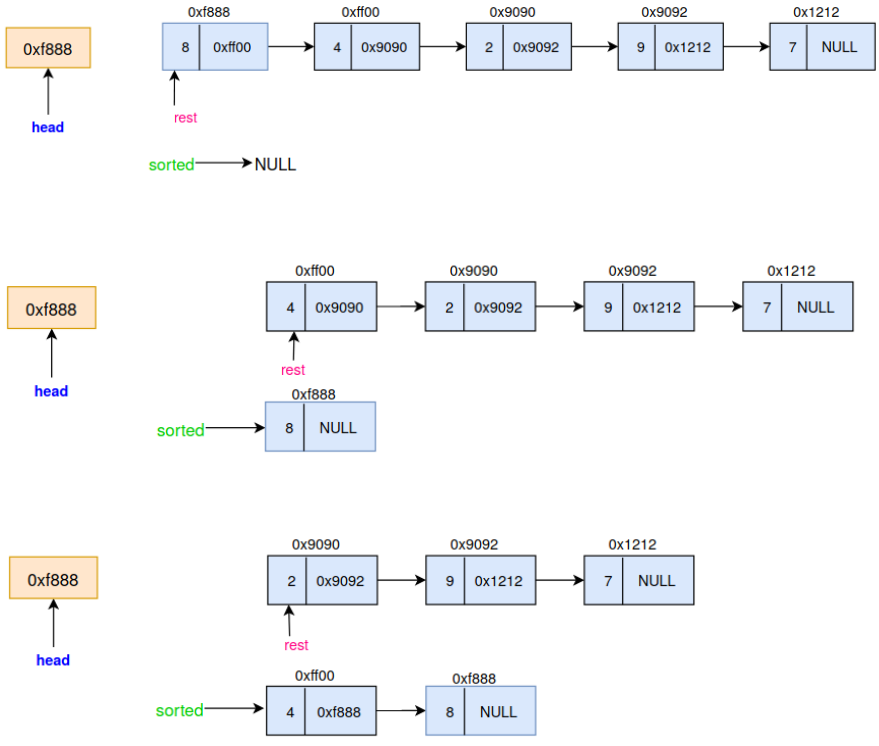
```
   else
   {
      // locate the node before the point of insertion
      auto current = *head;
      while (current->next!=NULL && current->next->data <
   newNode->data)
      {
         current = current->next;
      }
      newNode->next = current->next;
      current->next = newNode;
   }
}
```

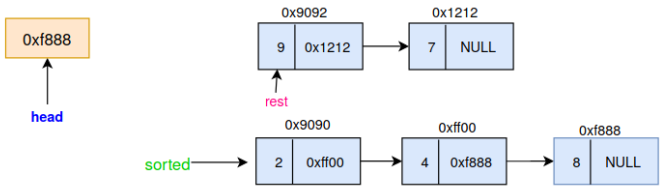


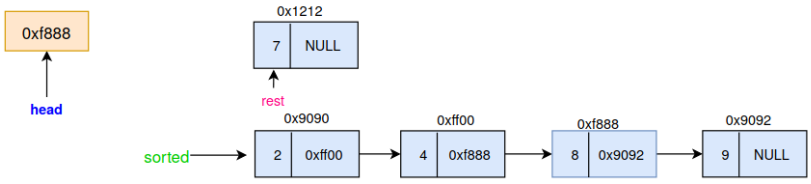


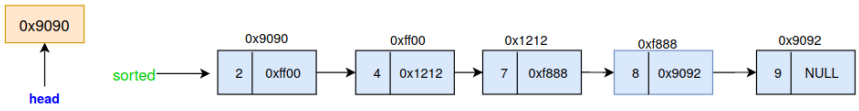


9. Тук идеята е, че пазим 2 списъка - един, в който вмъкваме елементите и той е винаги сортиран (*sorted*) и останалата част, тоест частта от оригиналния списък, която все още не е вмъкната, от нея вземаме възлите, които ще добавяме в сортираната. Продължаваме вече няма елементи във втория списък. Връщаме *sorted*, тоест насочваме *head* към него.





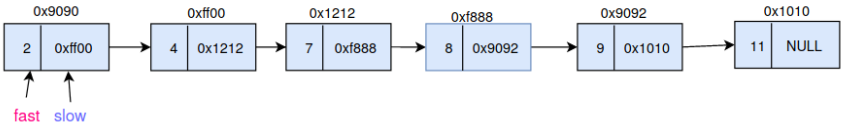


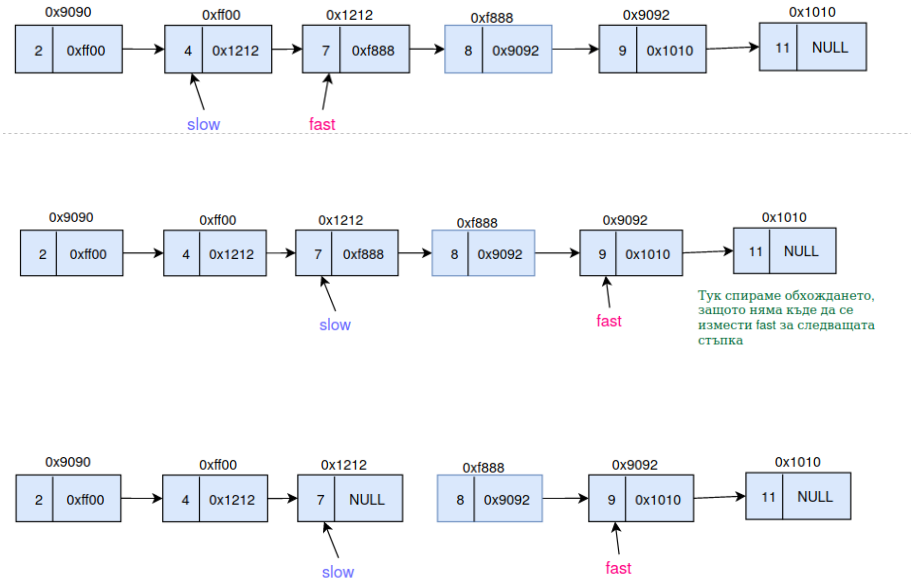



```
template <typename T>
void insertionSort(Node<T> ** head)
{
   Node<T> * sorted = nullptr, *rest = *head;
   while (rest != nullptr)
   {
      Node<T> *temp = rest;
      rest = rest->next;
      temp->next = nullptr; // set to null, as to "leave"
   the other list (it is now not pointing to any element
   from it)
      sortedInsert(&sorted, temp);
   }
   *head = sorted;
}
```

10. Най-простото решение, което ни идва на ум е първо да намерим дължината на списъка, да я разделим на две и с второ обхождане да направим делението. Тук ще използваме един малко по-хитър подход: ще пазим 2 указателя, с които да итерираме: един *slow*, който по стандартен начин минава елемент по елемент и един *fast*, който обхожда списъка през елемент. Идеята е, че когато *fast* достигне края на списък, то *slow* ще е в средата и ще можем да извършим делението там.

```
template <typename T>
void frontBackSplit(Node<T> ** head, Node<T> ** front,
   Node<T> ** back)
{
   if (*head == nullptr)
   {
      *front = *back = nullptr;
      return;
   }
   Node<T> *slow = *head, *fast = *head;
   while (fast != nullptr && fast->next != nullptr && fast
   ->next->next != nullptr)
   {
      slow = slow->next;
      fast = fast->next->next;
   }
   *front = *head;
   *back = slow->next;
   slow->next = nullptr;
}
```





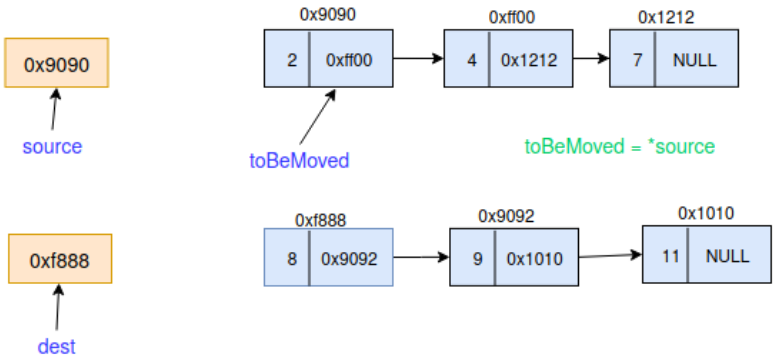
11. Решението на тази задача се състои в следното: на всяка стъпка гледаме текущия и следващия възел, ако са равни, то премахваме единия - втория, защото е по-лесно. Ако не са равни, то просто изместваме указателя напред.

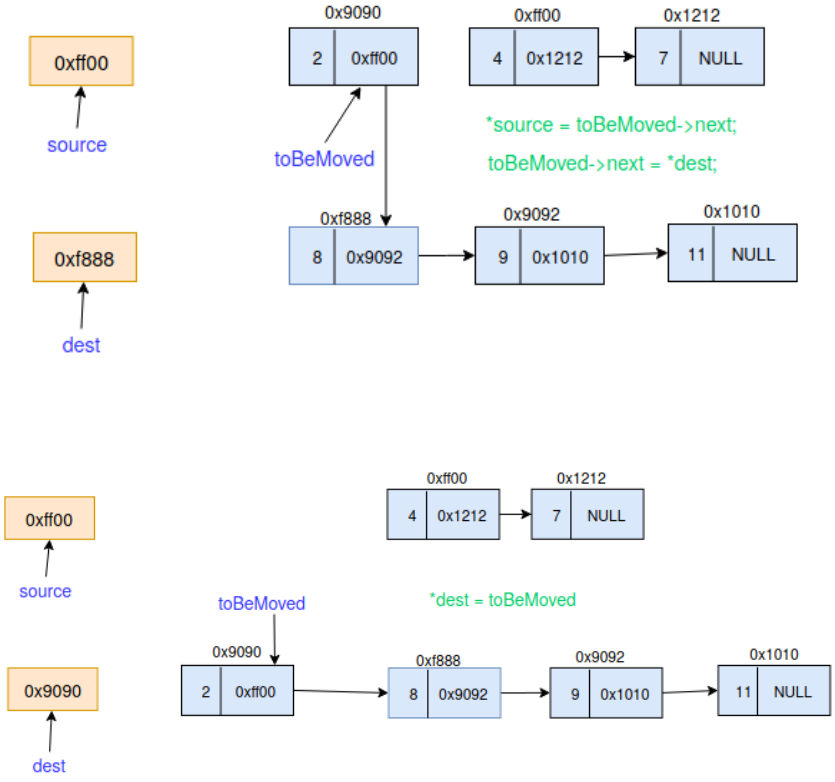
```
template <typename T>
void removeDuplicates(Node<T> ** head)
{
   Node<T> *current = *head, *prev = nullptr;
   if (current == nullptr) return;

   while (current->next != nullptr)
   {
      if (current->data == current->next->data)
      {
         Node<T> *next = current->next->next;
         delete next;
         current->next = next;
      }
      else
      {
```

```
            current = current->next;
        }
    }
}
```

12. Тук единственото, което може да малко е смяната на указателите. Вземаме първия възел от source, слагаме в началото на dest и преместваме source с едно напред.

```
void moveNode(Node<T> **dest, Node<T> **source)
{
    Node<T> *toBeMoved = *source;
    if (*source != nullptr) {
        Node<T> *toBeMoved = *source;
        *source = toBeMoved->next;
        toBeMoved->next = *dest;
        *dest = toBeMoved;
    }
    return;
}
```

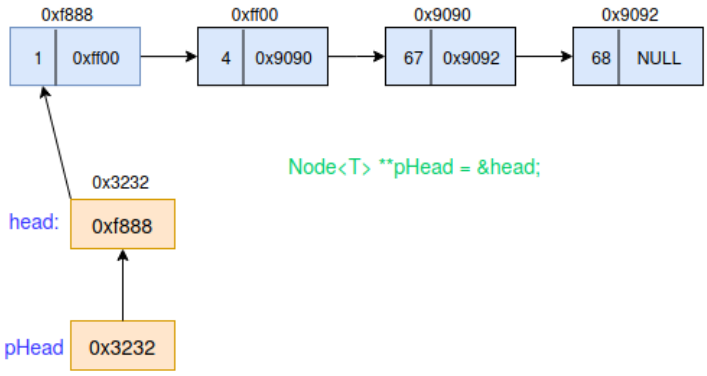




13. За да решим задачата минаваме последователно през всеки елемент от оригиналния списък, като редуваме добавяне в края в a и добавяне в края в b. За добавянето ще използваме функцията moveNode. Първоначално може да се запитате как ще я използваме за добавяне в края след като тя по своята дефиниция добавя елемент в началото *dest*. Тънкостта тук е първо в дефиницията на функцията:

```
void moveNode(Node<T> **dest, Node<T> **source)
```

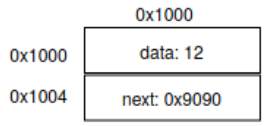
Виждаме, че тя приема двойни указатели, тоест указатели към клетки от паметта, в които за записани адреси на възли. Например, ако имаме списъка:



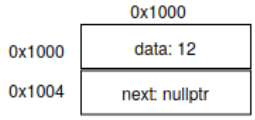
Мислим си за head като за клетка от паметта, в която е записан адресът на първия възел от списъка. *head* също има адрес, *pHead* съдържа именно този адрес. При извикване на *moveNode* ние просто променяме стойността на *head*. Нека разгледаме случай, в който head има записана стойността *NULL* и извикваме moveNode за pHead, като искаме да добавим възела вдясно:

Съгласно имплементацията на *moveNode* ще получим следното:

Тоест адресът, записан в *head* се променя, имаме вмъкване в празен списък. Това е интуитивно. Нека сега разгледаме подробно как изглежда един възел:



Да предположим, че *T=int*. Той си има адрес (*0x1000*), има си данни и клетка, в която е записан адресът на следващия възел. Тази клетка можем да използваме по аналогия на горния случай. Адресът ѝ е 0x1004 (забележете отместването от 4байта == размера на int), тоест съвсем спокойно можем да променяме нейната стойност с *moveNode*.
Сега нека разгледаме последния възел от даден списък. Той би изглеждал така:



Вече би трябвало да е очевидно, как се добавя в края на списъка - достатъчно е да подаден на moveNode адреса на next на последната клетка. Тоест нещо от рода на:

```
moveNode(&current->next, &node)
```

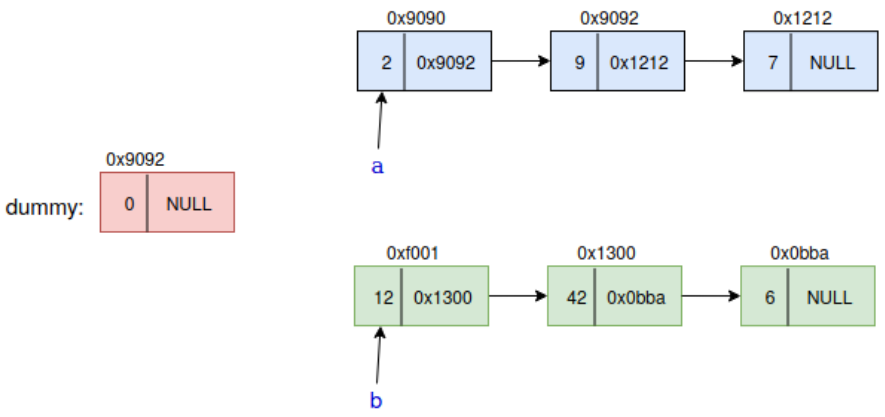
Да се върнем към решението на задачата. То се състои в просто в обхождане на списъка и добавяне последователно в края на единия списък и в края на другия списък. Една важна особеност е, че се правим два dummy възела. Правим ги просто за да използваме тяхната *next* клетка и по-точно нейния адрес, за да можем да викаме *moveNode* в нея, тоест да добавяме в края на списъка.

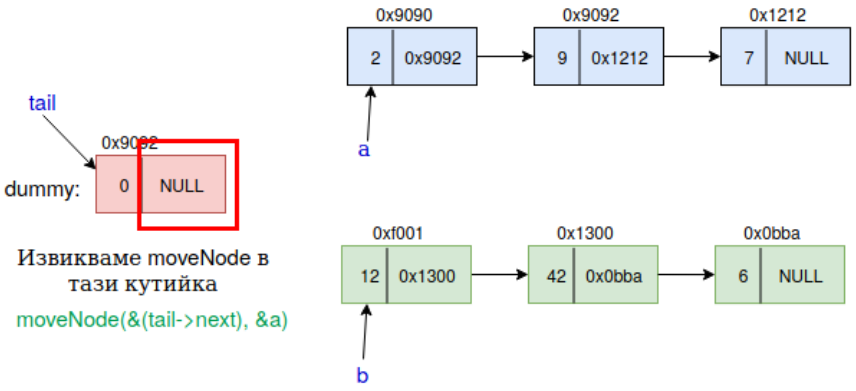
```
template <typename T>
void alternatingSplit(Node<T> **head, Node<T> **aList,
   Node<T> **bList)
{
   Node<T> aDummy, bDummy; // default values
   Node<T> *aTail = &aDummy, *bTail = &bDummy;
   auto current = *head;

   while (current != nullptr)
   {
      moveNode(&(aTail->next), &current);
      aTail = aTail->next;
      if (current != nullptr)
      {
         moveNode(&(bTail->next), &current);
         bTail = bTail->next;
      }
   }

   *aList = aDummy.next;
   *bList = bDummy.next;
}
```

14. Решението на тази задача е много подобно на това на предната, но добавяме само в един списък. Първото решение, което ще покажем е отново с dummy възел и логиката е като по-горе - просто накрая връщаме следващия го възел.

```
template <typename T>
Node<T> *shuffleMerge(Node<T> *a, Node<T> *b)
{
   Node<T> dummy(6);
   Node<T> *tail = &dummy;

   while (true)
   {
      if (a == nullptr)
```

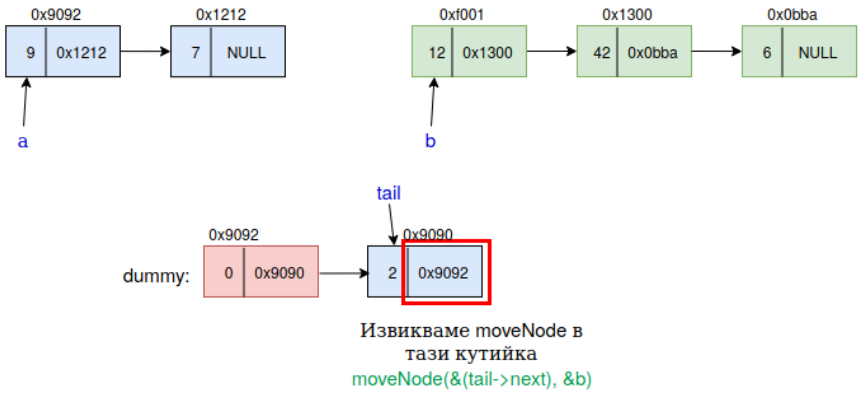
```
        {
            // b is empty
            tail->next = b;
            break;
        }
        else if (b == nullptr)
        {
            // a is empty
            tail->next = a;
            break;
        }
        else
        {
            moveNode(&(tail->next), &a);
            tail = tail->next;
            moveNode(&(tail->next), &b);
            tail = tail->next;
        }
    }

    return dummy.next;
}
```





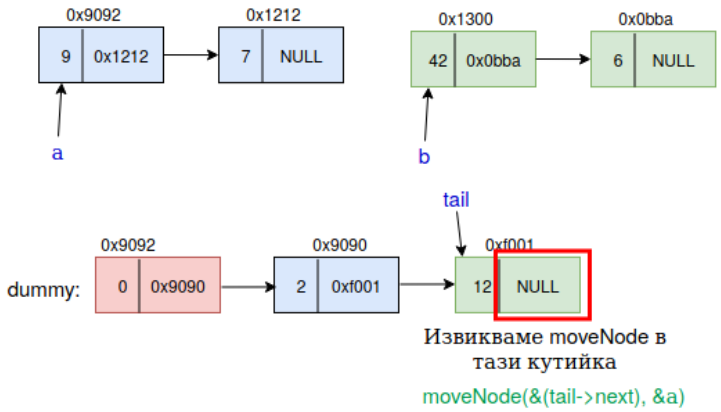
Извикваме moveNode в тази кутийка

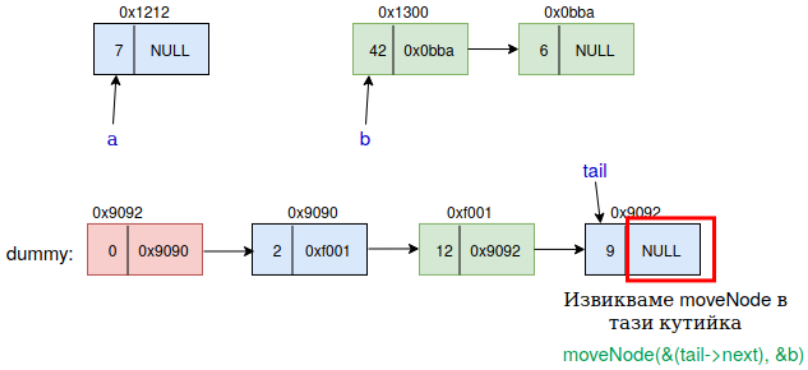
moveNode(&(tail->next), &a)

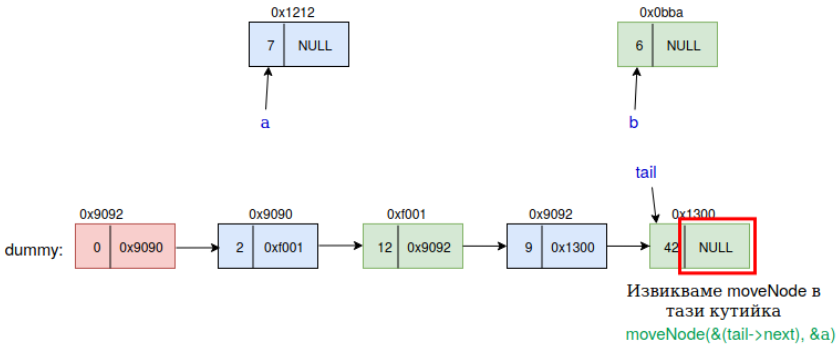


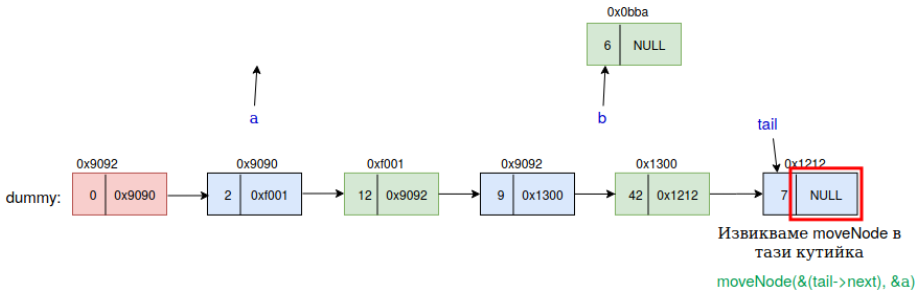
Извикваме moveNode в тази кутийка

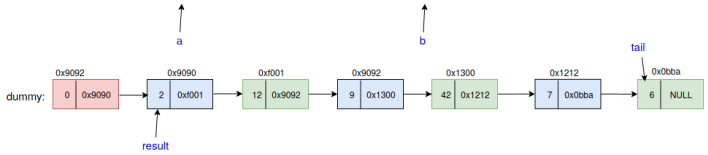
moveNode(&(tail->next), &b)


0x9092
9
0x1212
0x1212
7
NULL
0x1300
42
0x0bba
0x0bba
6
NULL
a
b
tail
dummy:
0x9092
0
0x9090
0x9090
2
0xf001
0xf001
12
NULL
Извикваме moveNode в тази кутийка
moveNode(&(tail->next), &a)


0x1212
7
NULL
0x1300
42
0x0bba
0x0bba
6
NULL
a
b
tail
dummy:
0x9092
0
0x9090
0x9090
2
0xf001
0xf001
12
0x9092
0x9092
9
NULL
Извикваме moveNode в тази кутийка
moveNode(&(tail->next), &b)


0x1212
7
NULL
0x0bba
6
NULL
a
b
tail
dummy:
0x9092
0
0x9090
0x9090
2
0xf001
0xf001
12
0x9092
0x9092
9
0x1300
0x1300
42
NULL
Извикваме moveNode в тази кутийка
moveNode(&(tail->next), &a)



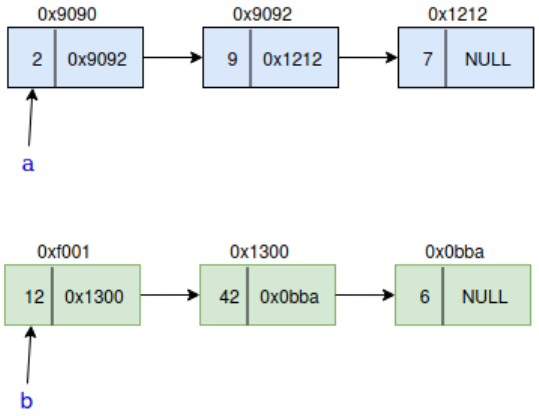


Тъй като използваме dummy възела само заради неговата next клетка, то може просто да си направим клетка, която е празна и двоен указател, който пази адреса ѝ, като съответно след това работим само с двойния указател.

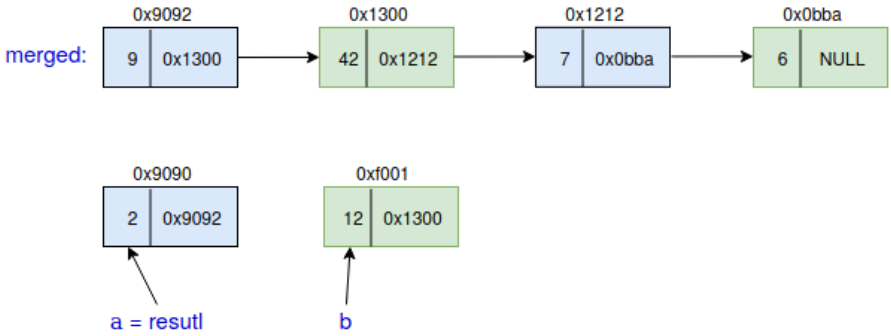
```
template <typename T>
Node<T> *shuffleMerge(Node<T> *a, Node<T> *b)
{
   Node<T> *result = nullptr;
   Node<T> **lastPtr = &result;

   while (true)
   {
      if (a == nullptr)
      {
         // b is empty
         *lastPtr = a;
         break;
      }
      else if (b == nullptr)
      {
         // a is empty
```

```
            *lastPtr = b;
            break;
        }
        else
        {
            moveNode(lastPtr, &a);
            lastPtr = &((*lastPtr)->next);
            moveNode(lastPtr, &b);
            lastPtr = &((*lastPtr)->next);
        }
    }

    return result;
}
```

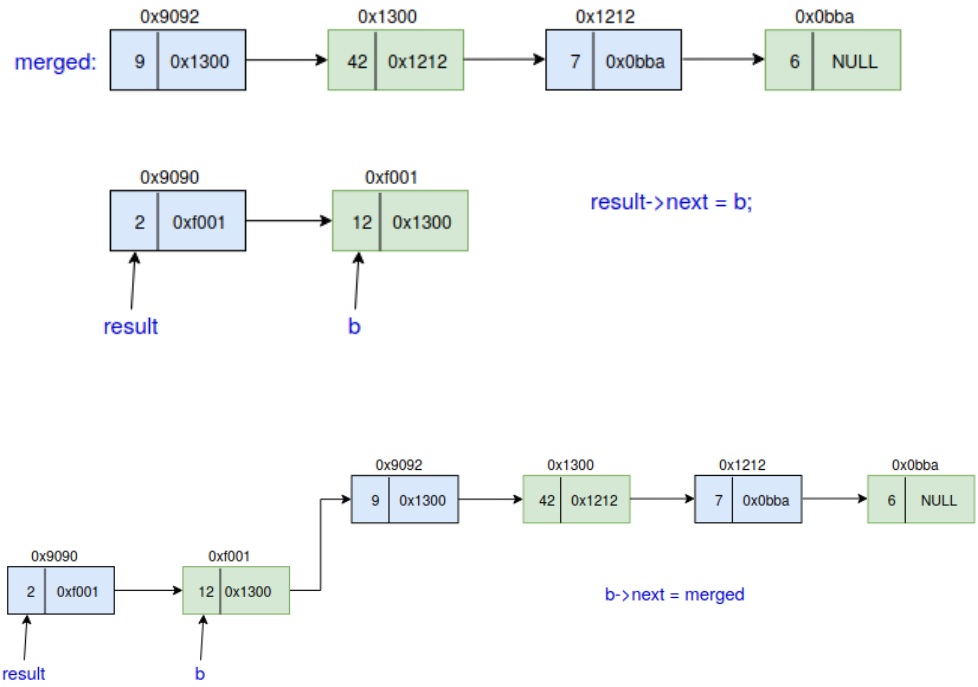
За рекурсивното решение нека разгледаме един пример. Нека са ни дадени 2 списъка, които бихме искали да слеем:



На всяка стъпка имаме достъп до първите елементи от двата списъка (списъците са представени чрез тях). Нека вземем тези два елемента и си представим какъв би бил резултатът от рекурсивното извикване на функцията над останалите части от списъците - то си е просто сливането им, тоест изглежда по такъв начин:



Остава ни само да решим как да добавим първите елементи, които сме си запазили. Ами просто ги добавяме отпред, първо a, после b, а след b ще залепим вече слятата част.



Разбира се не трябва да забравяме и за случаите, в които единият от двата списъка е празен. Тогава автоматично връщаме другия.

```
Node<T> *shuffleMergeReq(Node<T> *a, Node<T> *b)
{
   if (a == nullptr)
   {
      return b;
   }
   if (b == nullptr)
   {
      return a;
   }
   Node<T> *result = nullptr;
   Node<T> *merged = sortedMergeRec(a->next, b->next);
   result = a;
   result->next = b;
   b->next = merged; // a -> b -> merged
   return result;
}
```

15. Решението на тази задача много наподобява това на предната, само с една разлика - на всяка стъпка добавяме само по един възел към резултата, а именно по-малкия от първите възли на a и b. Отново имаме няколко варианта за решение. Първият е с dummy възел както по-горе:

```
void sortedMerge(Node<T> **merged, Node<T> *&a,
   Node<T> *&b) {
   Node<T> dummy{6};
   Node<T> *tail = &dummy;

   while (true)
   {
      if (b == nullptr)
      {
         // b is empty
         tail->next = a;
         break;
      }
```

```
        else if (a == nullptr)
        {
            // a is empty
            tail->next = b;
            break;
        }
        else
        {
            if (a->data <= b->data)
            {
                moveNode(&(tail->next), &a);
                tail = tail->next;
            }
            else
            {
                moveNode(&(tail->next), &b);
                tail = tail->next;
            }
        }
    }

    *merged = dummy.next;
}
```

Тук отново *tail* сочи винаги към последния елемент на списъка. За рекурсивното решение следваме логиката от по-горе, а именно гледаме първите елементи на двата списъка и запазваме по-малкия. След това рекурсивно извикваме, но гледаме остатъка на списъка, чийто елемент е по-малък, а другия списък го разглеждаме без да премахваме нищо от него.

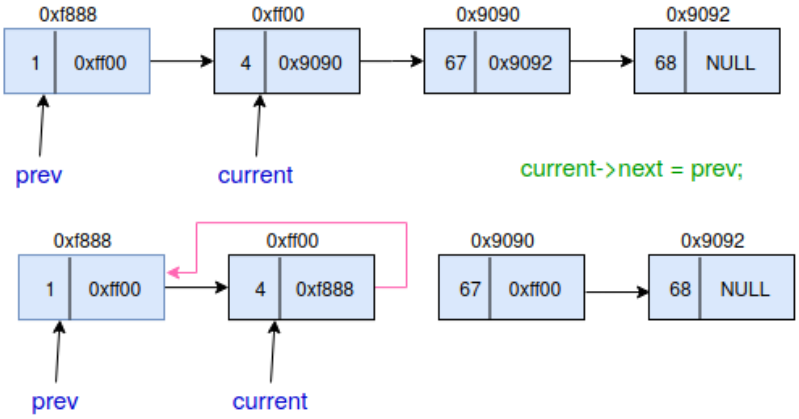
```
Node<T>* mergeReq(Node<T> * a, Node<T> * b)
{
    Node<T> *result = nullptr;
    if (a == nullptr) {
        return b;
    } else if (b == nullptr) {
        return a;
```

```
    }
    if (a->data <= b->data) {
        result = a;
        result->next = mergeReq(a->next, b);
    } else {
        result = b;
        result->next = mergeReq(a, b->next);
    }
    return result;
}
```

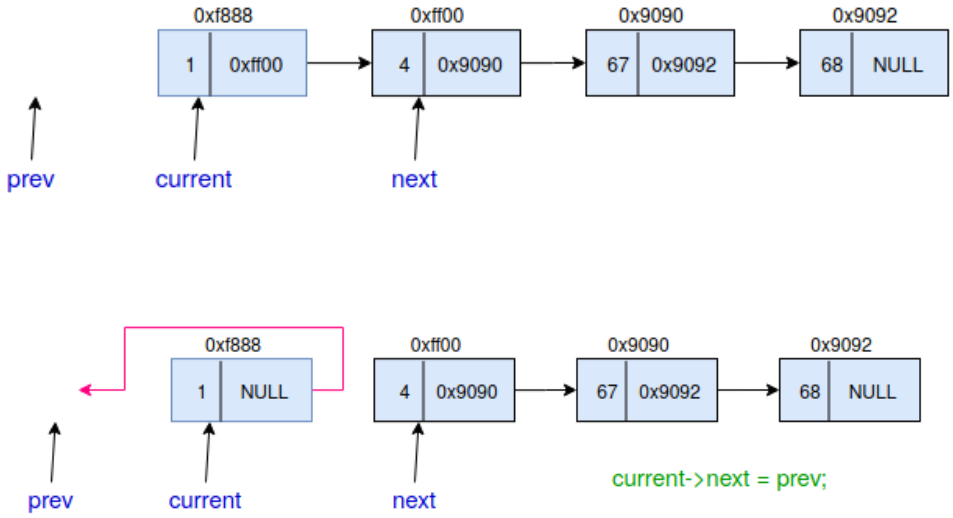
16. Имаме всичко, което ни трябва за да имплементираме алгоритъма. За разделянето в 2 части ще използваме *frontBackSplit*, а за сливането на вече сортираните части ще използваме *sortedMerge*.

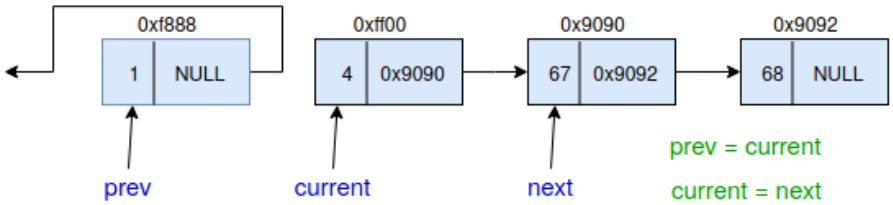
```
void mergeSort(Node<T> **head)
{
    if (*head == nullptr ||  (*head && (*head)->next ==
    nullptr))
    {
        // bottom;
        return;
    }

    Node<T> *a, *b;
    frontBackSplit(head, &a, &b);
    mergeSort(&a);
    mergeSort(&b);
    sortedMerge(head, a, b);
}
```

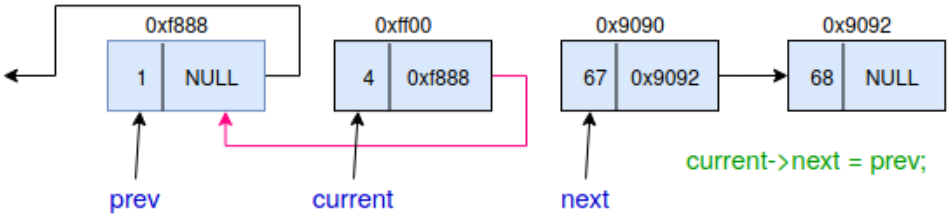
17. **Итеративно решение:** преминаваме през всеки възел и променяме указателя *next* да сочи към предходния възел. Моментално ни идва на ум, че трябва по някакъв начин да пазим предния възел. И нека видим едно такова изместване:

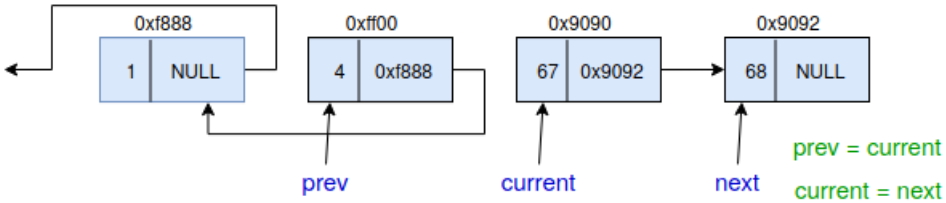


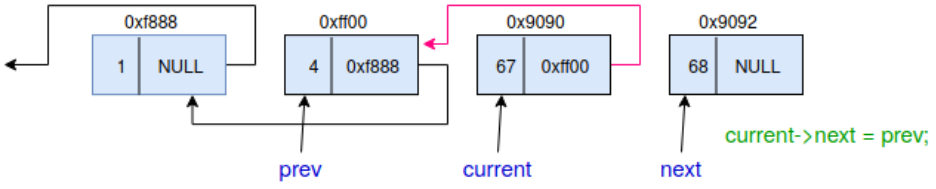
Виждаме, че така няма как от *current* да стигнем до следващия го елемент, тоест трябва да запазим следващия на *current* преди да извършим промяната. Нагледно:

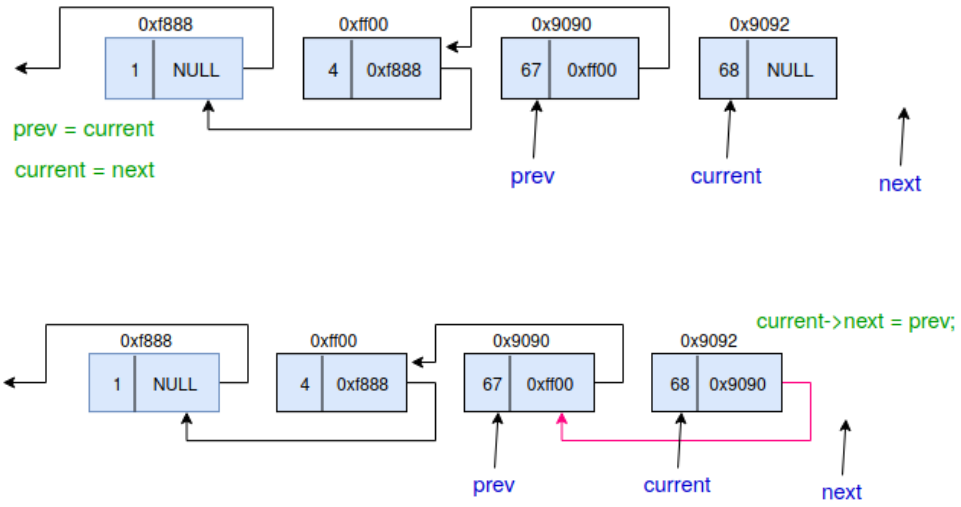



0xf888
1
NULL
0xff00
4
0x9090
0x9090
67
0x9092
0x9092
68
NULL
prev
current
next
prev = current
current = next

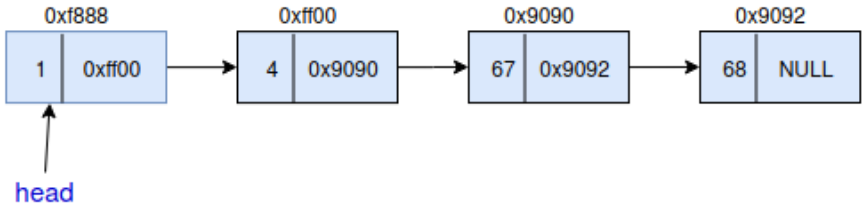

0xf888
1
NULL
0xff00
4
0xf888
0x9090
67
0x9092
0x9092
68
NULL
prev
current
next
current->next = prev;


0xf888
1
NULL
0xff00
4
0xf888
0x9090
67
0x9092
0x9092
68
NULL
prev
current
next
prev = current
current = next

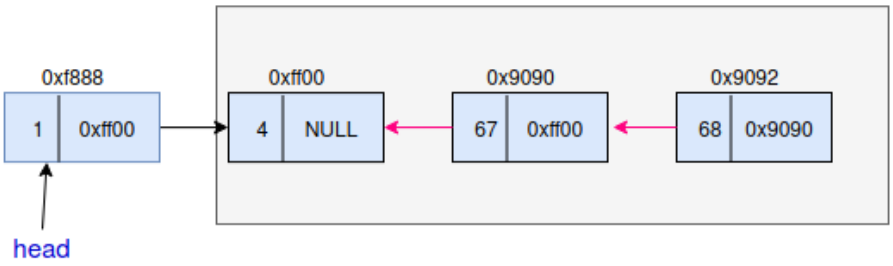

0xf888
1
NULL
0xff00
4
0xf888
0x9090
67
0xff00
0x9092
68
NULL
prev
current
next
current->next = prev;



Рекурсивно решение: тук нека отново разгледаме пример. Нека имаме свързан списък:

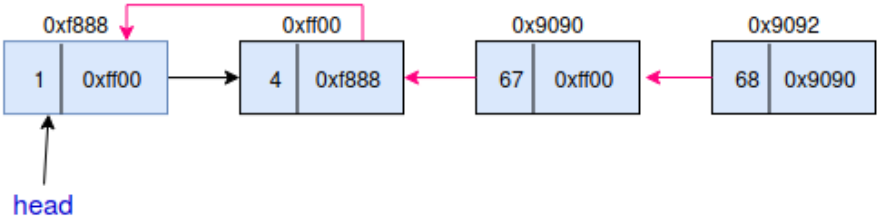


И нека си представим резултата, ако сме извикали рекурсивно функцията над списъка без първия елемент, тоест нещо от рода на reverse(&(head->next)):



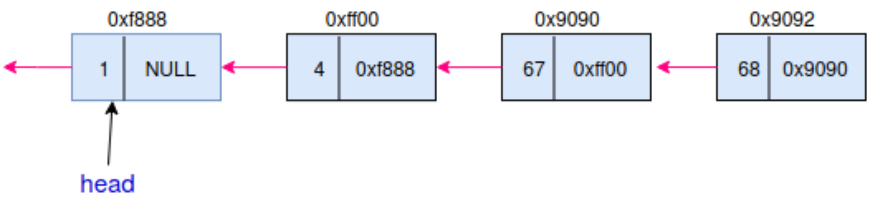
Остава ни само да насочим втория възел (с адрес 0xff00) да сочи към текущия и текущия да стане последен (защото в момента е първи). Първото правим по следния начин:

```
head->next->next = head;
```



А второто така:

```
head->next = nullptr;
```



И това е решението. Разбира се трябва да помислим и за дъното на това рекурсивно извикване, а то е именно когато имаме един елемент или празен списък.

```
void reverseReq(Node<T> **head)
{
   if (*head == nullptr || (*head != nullptr && (*head)->
   next == nullptr)) return;

   Node<T> *first = *head;
   Node<T> *rest = (*head)->next;

   reverseReq(&rest);

   first->next->next = first;
   first->next = nullptr;

   *head = rest;
}
```