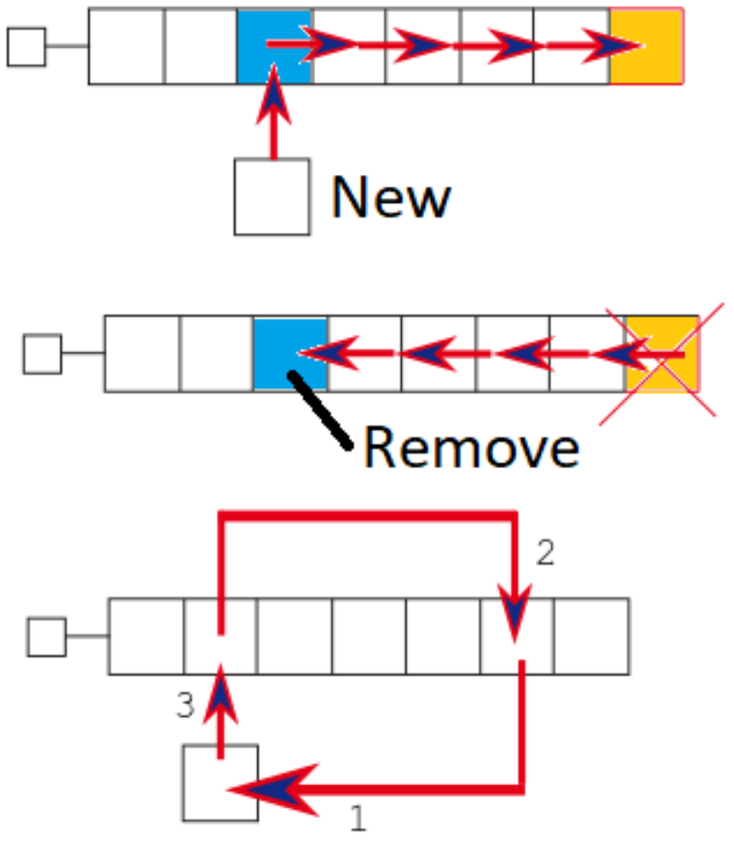
Linear data structures (1)

In linear data structures the elements (integers, pointers, structs, etc.) are ordered – i.e. we may say, which member is the first, which is the seconds, etc. The simplest ordered linear data structure is array.



Problems with array:

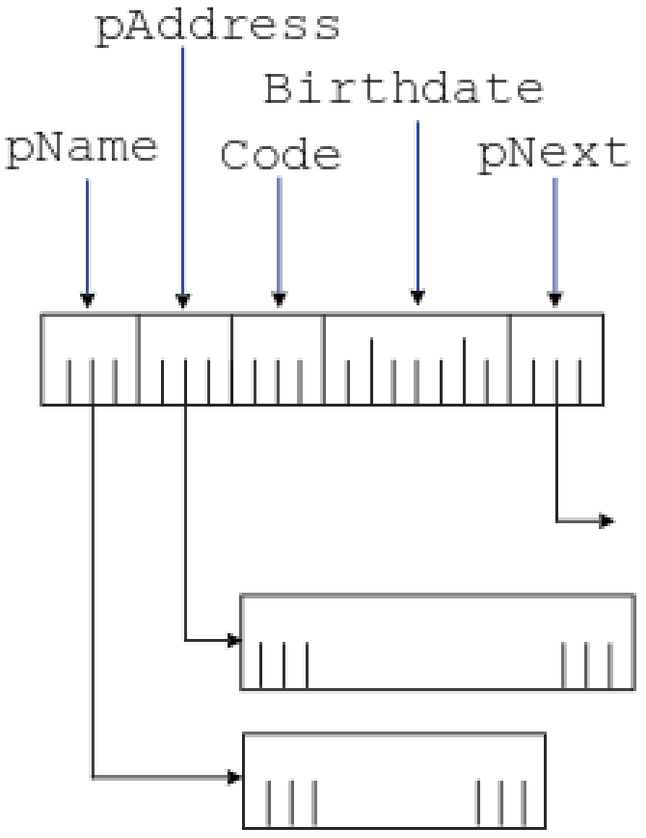
1. If we need to insert a new element, we must at first check, is there at the end of array some free space. If not, we have to reallocate our memory field. Often it is accompanied with relocating of large amounts of data. After that we need to free the position for the new element: i.e. once more shift data.
2. If we need to remove an element, we must shift data to left to cover the position. One position at the end of array becomes unused. There is an alternative solution: do not shift but somehow mark that the position as empty (for example fill with zeroes).

Linear data structures (2)

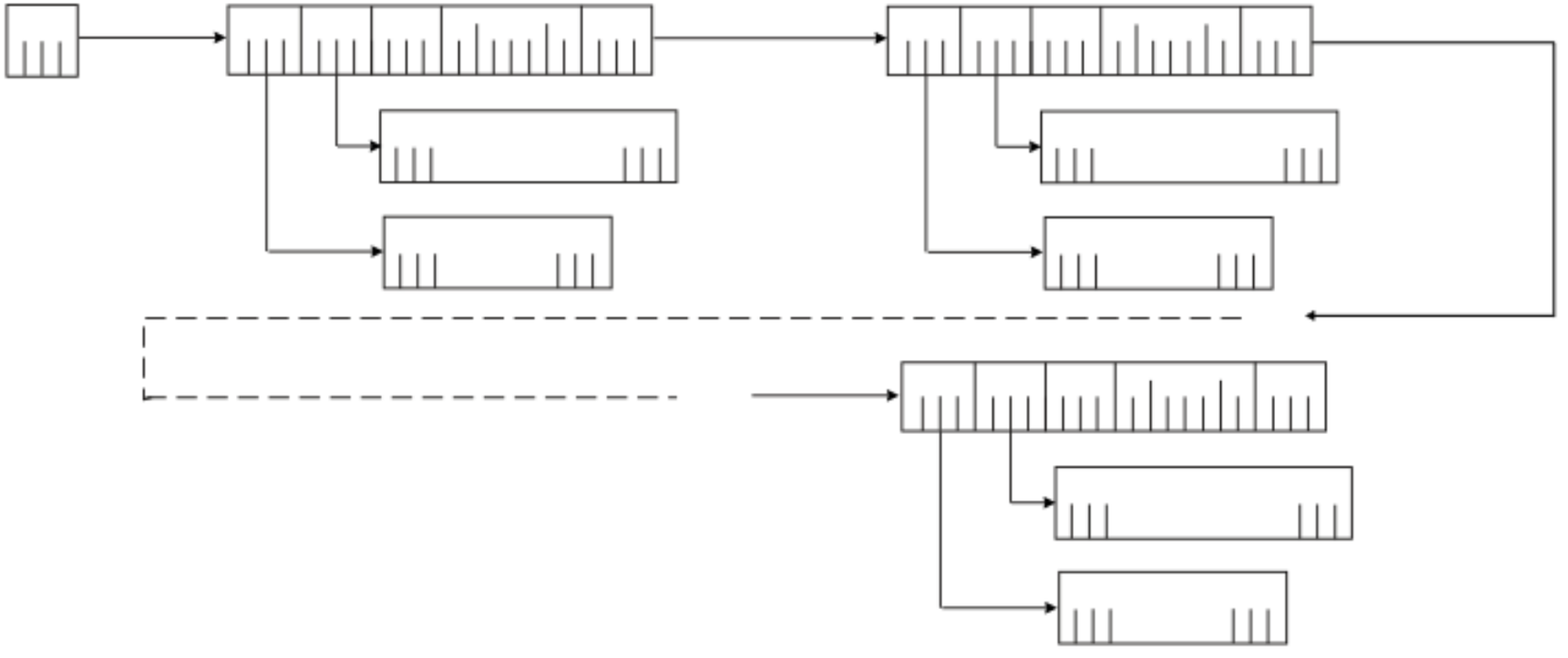
Let us have:

```
struct date
{
  short int Day = 0;
  char Month[4] = { }; // like "Jan", "Feb", etc.
  short int Year = 0;
};
typedef struct date DATE;

struct person
{
  char *pName = nullptr,
       *pAddress = nullptr;
  long int Code = 0;
  DATE Birthdate;
  person *pNext = nullptr;
};
typedef struct person PERSON;
```



Linear data structures (3)

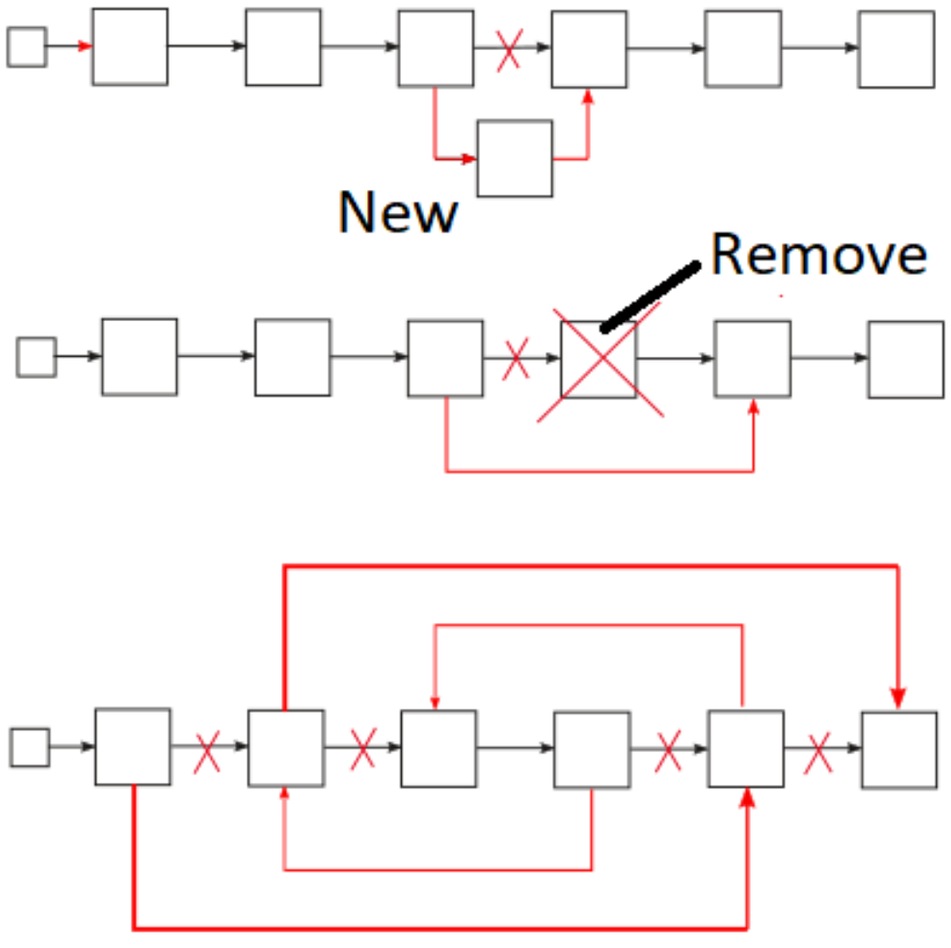


With pointer *pNext* we create linked list:
PERSON *pList; // points to the first element

Pointer *pNext* of the first element points to the second element, pointer *pNext* of the second element points to the third element, etc. Pointer *pNext* of the last element is zero.

The linked list does not need a long compact memory field. The elements may be in the heap higgledy-piggledy, without any order. But due to the pointers the data structure itself is ordered.

Linear data structures (4)

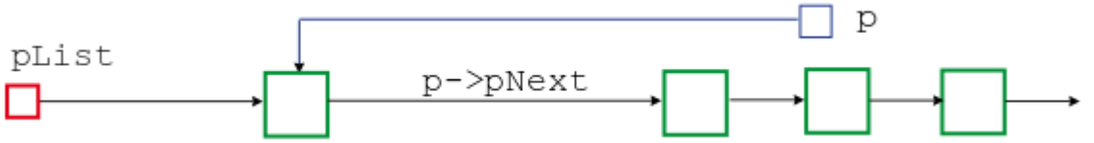


Inserting a new element and removing an existing element is much more effective than those operations with arrays. We do not need to shift large amounts of data and all the elements of list keep their current location. The only task we need to perform is to reset the *pNext* pointers.

The disadvantage of linked list is that we cannot use indeces. To access the i-th element we have to move from the first element (this is the only element we can access directly) to second, from the second to the third, etc. In an array we need just write like **(pArray +i)* or *Array[i]*.

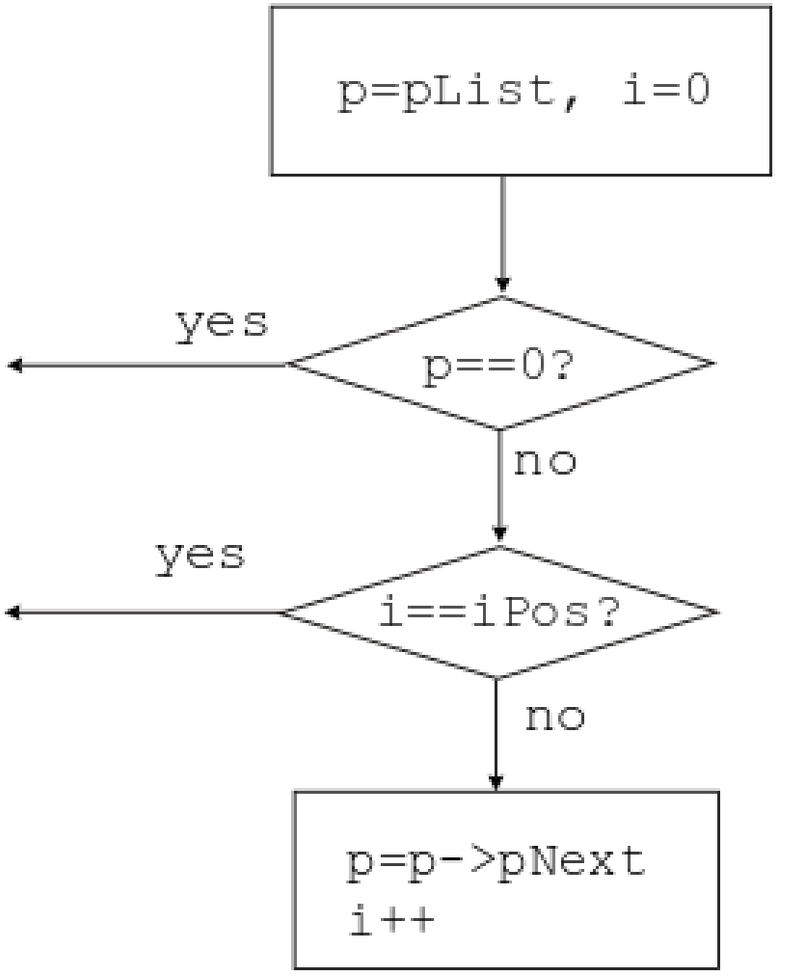
In data processing linked lists are the most used type of linear data structures. Arrays are used only when the amount of data is not large and the expected max number of data is well known. If the number of elements is unpredictable and continually changing, the linked lists have no alternatives.

Linear data structures (5)



Example: iteration through linked list

```
PERSON *GetPerson(PERSON *pList, int iPos)
{ // we want to get the pointer to item on position iPos
  if (!pList || iPos < 0) // check input
      return nullptr; // errors
  PERSON *p; // auxiliary variable
  int i; // auxiliary variable
  for (i = 0, p = pList;
        p && i < iPos;
        p = p->pNext, i++);
   return p;
}
```

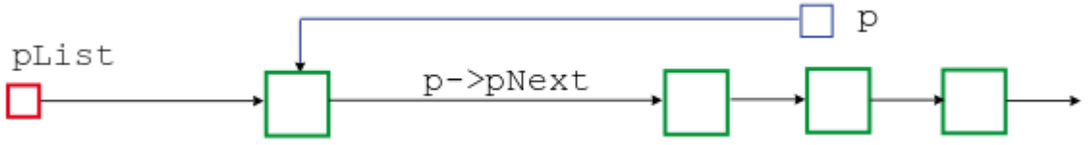


Linear data structures (6)

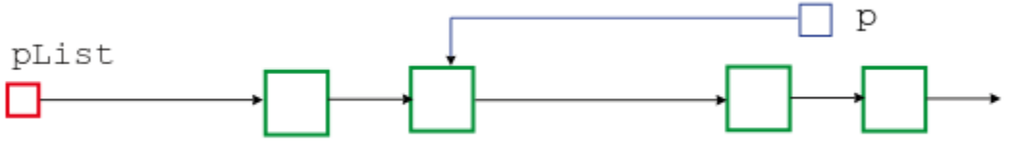
Suppose *iPos* is 2, i.e. we want to get the pointer to third item.

Loop starts: p = pList, i = 0;
p points to the first item.

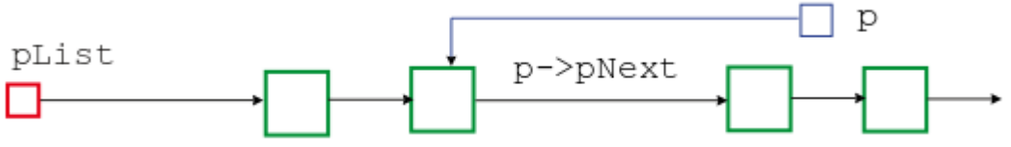
As *p* is not zero and *i* < *2*, looping continues. *p->pNext* is the address of second item.



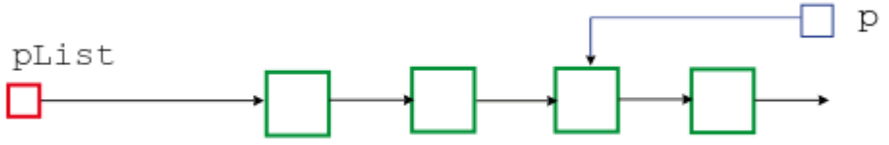
p = p->pNext, i++; After that *p* points to the second item and *i* is 1.



As *p* is not zero and *i* < *2*, looping continues. *p->pNext* is the address of third item.

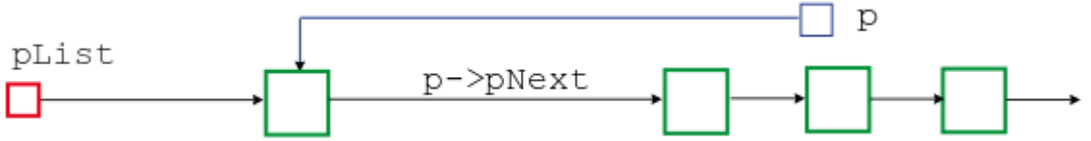


p = p->pNext, i++; After that *p* points to the third item and *i* is 2.



As *i* is now 2, the looping breaks off and we may return *p* as the searching result.

Linear data structures (7)



Example: iteration through linked list

```
PERSON *GetPerson(PERSON *pList, char *pKey)
{ // we want to get the pointer to person with name specified by the key
  if (!pList || !pKey) // check input
      return nullptr;
  PERSON *p; // auxiliary variable
  for (p = pList;  p && strcmp(pKey, p->pName); p = p->pNext);
  // strcmp() compares two strings. If they are identical, the return value is 0.
  // The iteration stops when p points to item with name identical with key or when
  // p is zero (i.e. the item we need does not exist)
   return p;
}
```

A key is something (string, integer, etc.) that we can directly or after some calculations retrieve from the record. Requirements: there must be algorithms with which we can assert that:

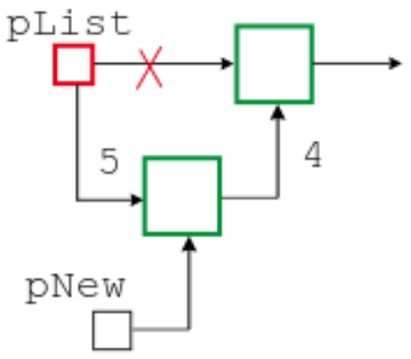
- two keys are equal
- if they are not equal, which of them is less

Linear data structures (8)

Example: insert into linked list

```
PERSON *Insert(PERSON *pList, PERSON *pNew, int iPos)
{ // we want to insert a new item into position iPos.
   // the function returns the pointer to first item
  if (!pNew || iPos < 0) // error in input
    return pList;
  if (!iPos)
 { // insert to the beginning, the new item will be the first one
    pNew->pNext = pList;
    return pNew;
  }
  PERSON *p; // auxiliary variable
  if (p = GetPerson(pList, iPos - 1))
 { // insert into the middle or to the end
    pNew->pNext = p->pNext;
    p->pNext = pNew;
  }
   return pList;
}
```

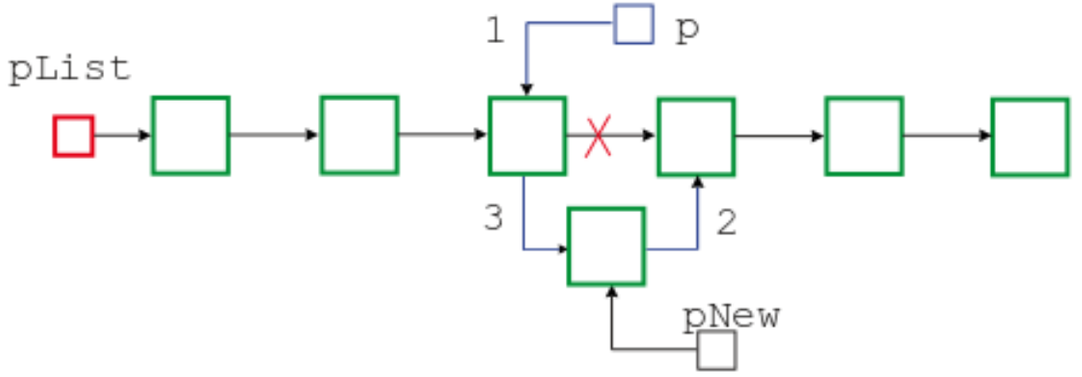
Linear data structures (9)




```
if (!iPos)
{ // insert to the beginning, the new item will be the first one
    pNew->pNext = pList; // 4
    return pNew; // 5
 }
```

On start *pList* points to the first item. As *iPos* is zero, the previous first item must be reduced to the second position. So the *pNext* member of the new item must point to the former first item (operation 4). The return value is the pointer to the new first item (operation 5).

Linear data structures (10)




```
if (p = GetPerson(pList, iPos - 1))// 1
{
   pNew->pNext = p->pNext; // 2
   p->pNext = pNew; // 3
 }
  return pList; // keeps its value
}
```

We want to insert the new item into position *iPos*. Consequently the item on position *iPos – 1* must start to point to the new item. Therefore the first thing to do is to find the pointer to item on position *iPos -1*. For that we may use function *GetPerson()* from slide *Linear data structures (5)* (operation 1). If *iPos* is wrong (negative or too large), *GetPerson()* returns 0 and the inserting will be omitted. If the item on position *iPos - 1* was found, we correct its *pNext* member (operation 3) and set the new item to point to item that was on position *iPos* and now is reduced to position *iPos +1* (operation 2).

Linear data structures (11)

Example: remove from linked list

```
PERSON *Remove(PERSON *pList, int iPos, PERSON **ppResult)
{ // we want to remove the item on position iPos
   // the removed item is not destroyed: the pointer to it is the output value
   // the function returns the pointer to first item
  if (!pList || iPos < 0 || !ppResult)
    return pList; // list is empty or errors in input data
 *ppResult = nullptr;
  PERSON *p; // auxiliary variable
  if (!iPos)
 {  // remove the first
   *ppResult = pList;
    pList = pList->pNext;
  }
  else if (p = GetPerson(pList, iPos - 1))
 {  // remove from the middle or from the end
     *ppResult = p->pNext;
     p->pNext = p->pNext->pNext;
  }
   return pList;
}
```

Linear data structures (12)

Usage example: we have linked list

```
PERSON *pStudentsGroup;
```

Remove the first and fourth students and print their names.

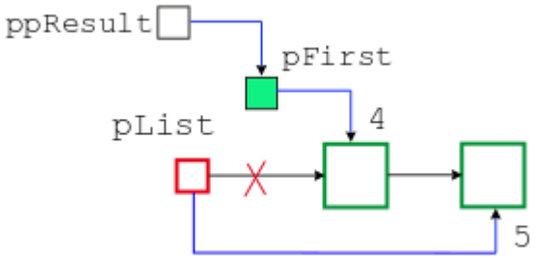
```
PERSON *pFirst, *pFourth;
pStudentGroup = Remove(pStudentGroup, 0, &pFirst);
if (pFirst)
{
  printf("Student %s was removed from list\n", pFirst->pName);
}
pStudentGroup = Remove(pStudentGroup, 4, &pFourth);
if (pFourth)
{
  printf("Student %s was removed from list\n", pFourth->pName);
}
```

On the last call to

```
PERSON *Remove(PERSON *pList, int iPos, PERSON **ppResult) { …… }
```

- the value of *pStudentsGroup* is copied into *pList*
- *iPos* gets value 4
- the pointer to *pFourth* (which itself is also a pointer) is calculated and copied into *ppResult*. In other words, *ppResult* will point to *pFourth*

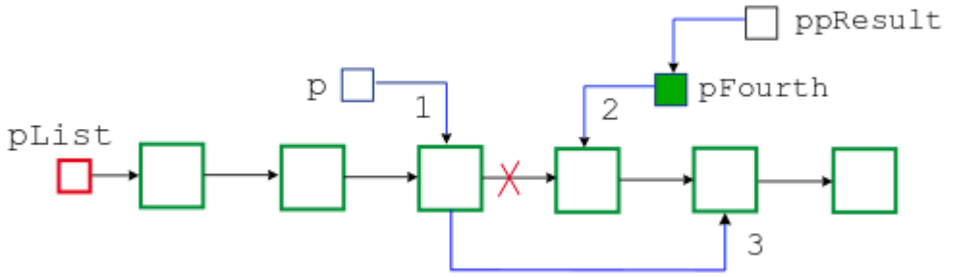
Linear data structures (13)




```
if (!iPos)
{  // remove the first
   *ppResult = pList; // 4
    pList = pList->pNext; // 5
}
```

The second item is now the first and *pList* must point to it (operation 5). To pointer *pFirst* (variable of the calling function and not the variable of *Remove()*) is assigned the pointer to the former first item (operation 4).

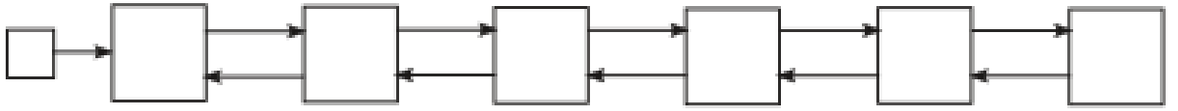
Linear data structures (14)




```
else if (p = GetPerson(pList, iPos - 1)) // 1
{
    *ppResult = p->pNext; // 2
    p->pNext = p->pNext->pNext; // 3
 }
  return pList;
}
```

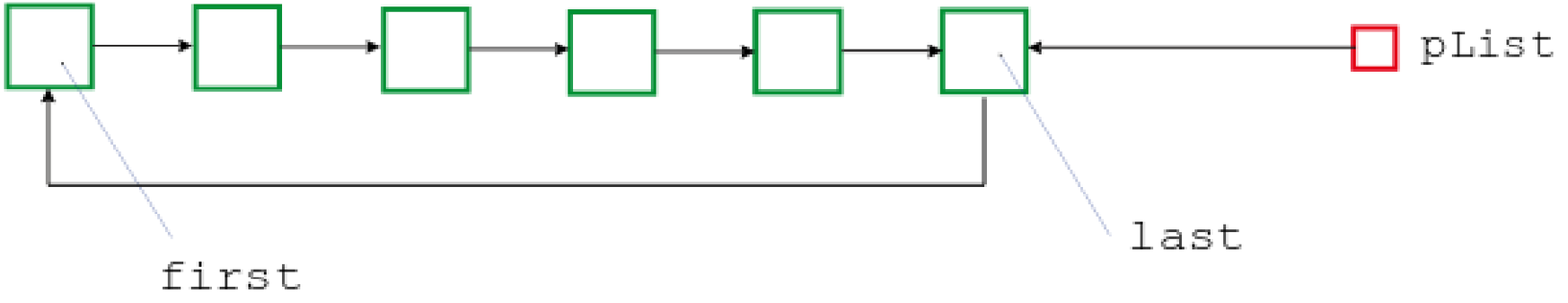
We want to remove the item on position *iPos*. Consequently the item on position *iPos – 1* must start to point to the item that is on position *iPos + 1*. Therefore the first thing to do is to find the pointer to item on position *iPos -1*. For that we may use function *GetPerson()* from slide *Linear data structures (5)* (operation 1). If *iPos* is wrong (negative or too large), *GetPerson()* returns 0 and the removing will be omitted. If the item on position *iPos - 1* was found, we correct its *pNext* member (operation 3). To pointer *pFourth* (variable of the calling function and not the variable of *Remove()*) is assigned the pointer to item that was on position *iPos* (operation 2).

Linear data structures (15)



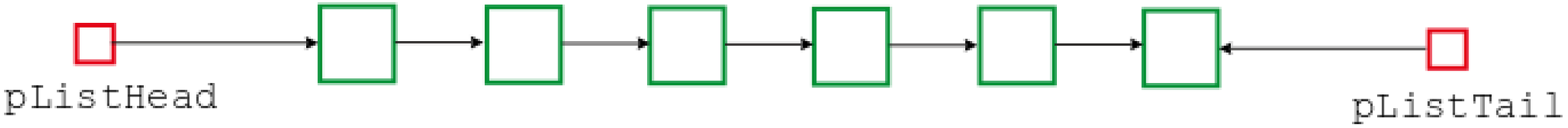
In double linked list we can move to both directions. Pointer *pPrior* in the first element is 0.

```
struct person
{
  char *pName = nullptr,
       *pAddress = nullptr;
  long int Code = 0;
  DATE Birthdate;
  struct Person *pNext = nullptr,
                *pPrior = nullptr;
};
```

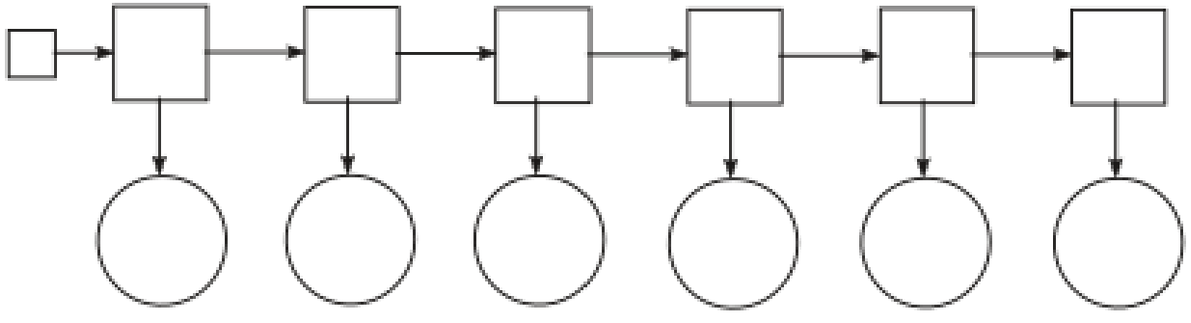


In circularly linked list the "last" element points to the "first" (terms "first" and "last" are conditional here).

Linear data structures (16)



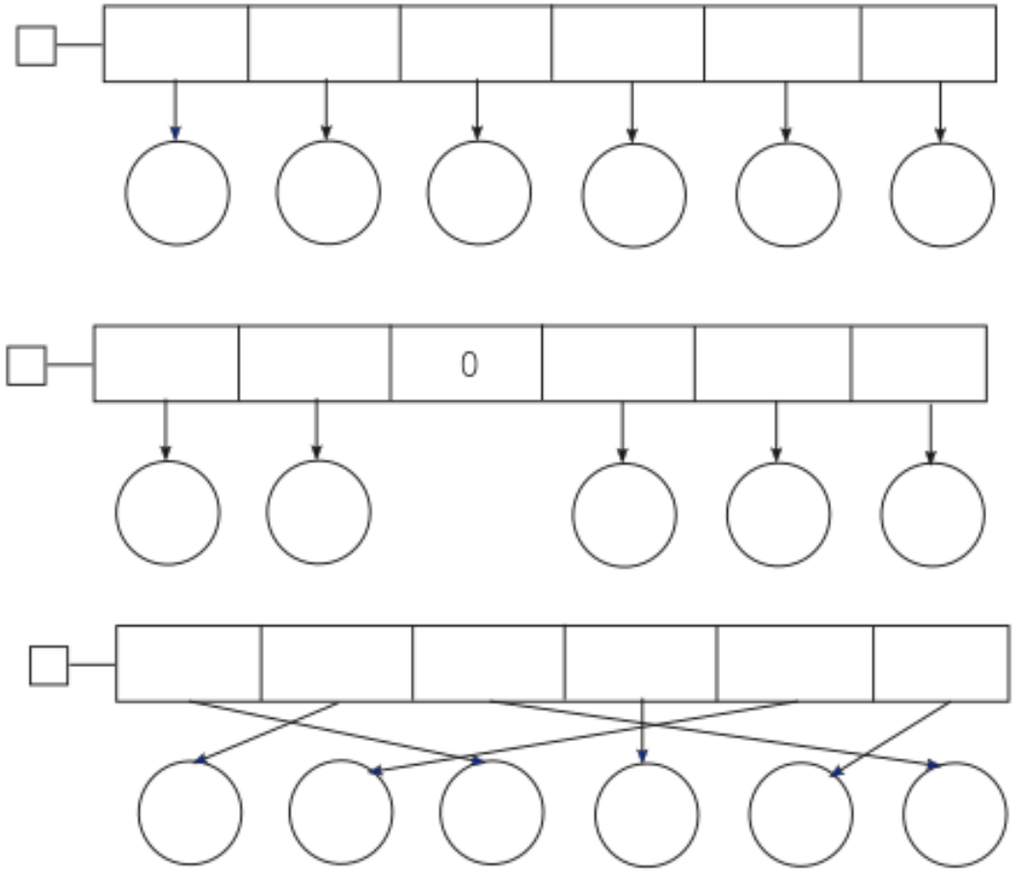
If the new elements must be always appended (and not inserted into the middle of list), it is useful to have 2 outside pointers: one to the first and one to the last element.



```
struct Header
{
  void *pRecord = nullptr;
   int type = 0;
   struct Header  *pNext = nullptr;
};
```

The separate headers are needed when the structs in data structure do not have *pNext* pointers or are of different types.

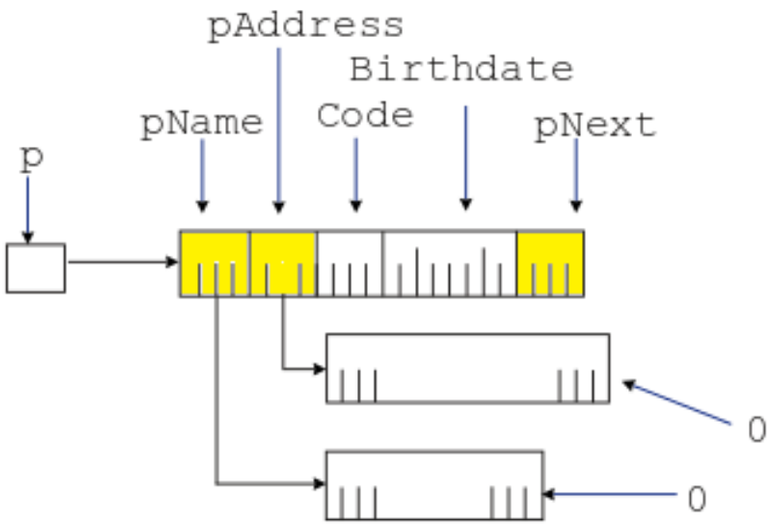
Linear data structures (17)

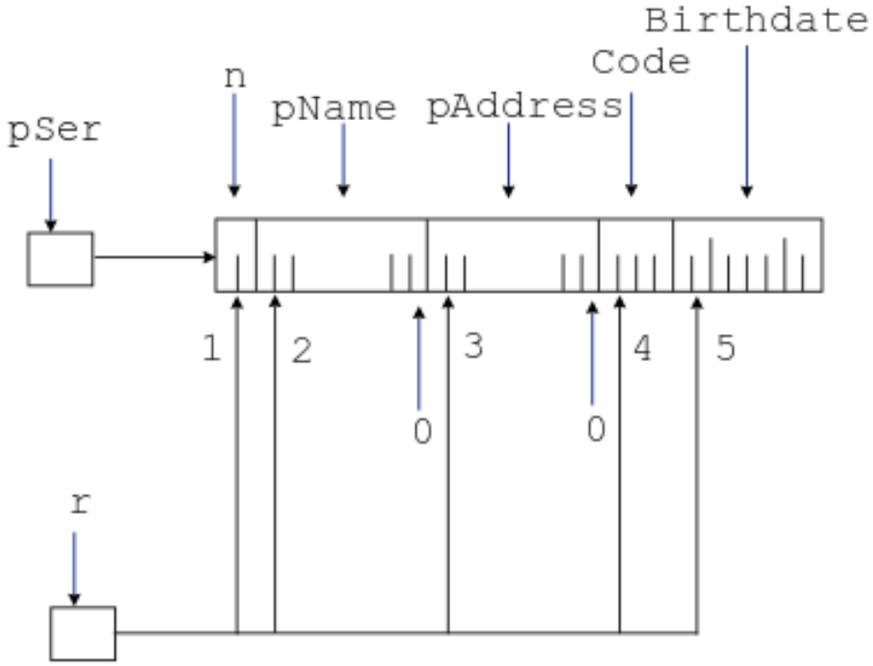



```
struct Header
{
    void *pRecord = nullptr;
    int type = 0;
};
```

This solution is very suitable but only if we are able the estimate the number of elements and thus allocate the vector with proper length. When deleting, instead of compressing simply replace the pointer with 0. When sorting, the structs are not moved because we may simply rearrange the pointers.

Serialization



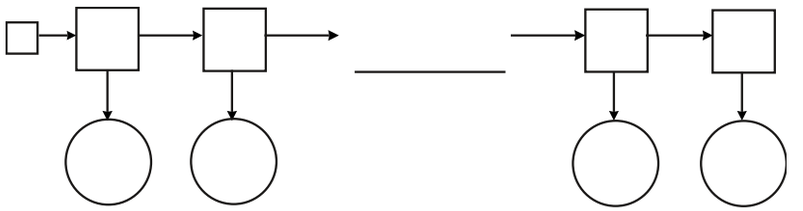


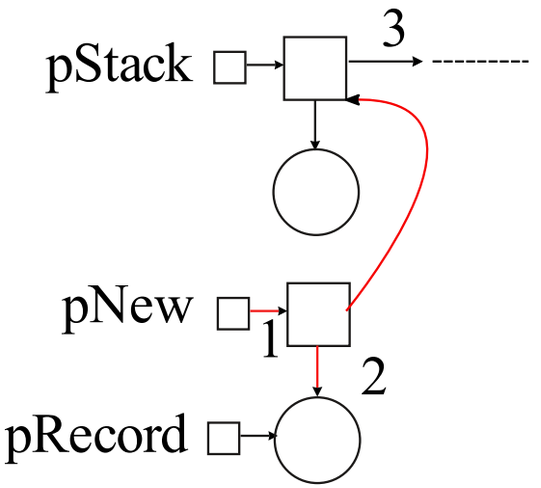

```
char *Serialize(PERSON *p) { // on disk memory addresses are senseless
  short int n1 = strlen(p->pName) +1, n2 = strlen(p->pAddress) + 1, n = n1 + n2;
  char *pSer, *r;
  pSer = new char [(n += sizeof(PERSON) + sizeof(int) – sizeof(PERSON *) -
                                2 * sizeof(char *)];
  memcpy(r=pSer,&n,sizeof(int)); //1
  memcpy(r+=sizeof(int),p->pName,n1); //2
  memcpy(r+=n1,p->pAddress,n2); //3
  memcpy(r+=n2,&p->Code,sizeof(long int)); //4
  memcpy(r+sizeof(long int),&p->Birthdate.day, sizeof(DATE)); //5
  return pSer; // serialized compact struct ready for writing to disk
}
```

Stack (1)

A stack or LIFO ("last in first out") is a list in which the insertions and deletions can be perfomed in only one fixed position, called the top. In implementations the top may be the first or the last element of list. In other words, the inserted record will be always on the top (operation push) and only the record on top (actually the most recently inserted record) can be removed (operation pop).

```
struct stack
{
      void *pRecord = nullptr;
      stack *pNext = nullptr;
};
stack *push(stack *pStack, void *pRecord)
{
    if (!pRecord)
    {
       throw invalid_argument ("No record to push");
    }
    stack *pNew = new stack;
    pNew->pRecord = pRecord;
    pNew->pNext = pStack;
    return *pNew;
}
```



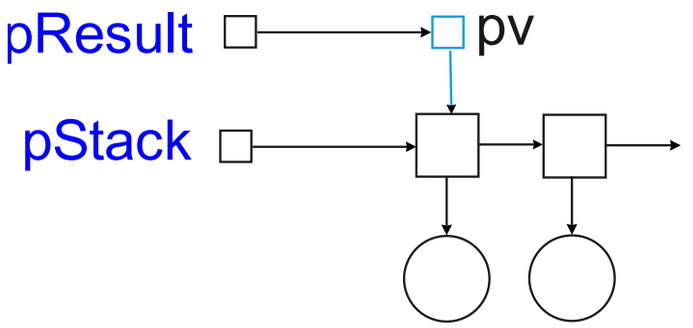


Stack (2)

```
stack *pop(stack *pStack, void **pResult)
{
      if (!pStack)
      {   // empty stack
           pResult = nullptr;
           return pStack;
       }
       *pResult = pStack->record;
       stack *p = pStack->pNext;
       delete pStack;
       return p;
}
```

Usage:

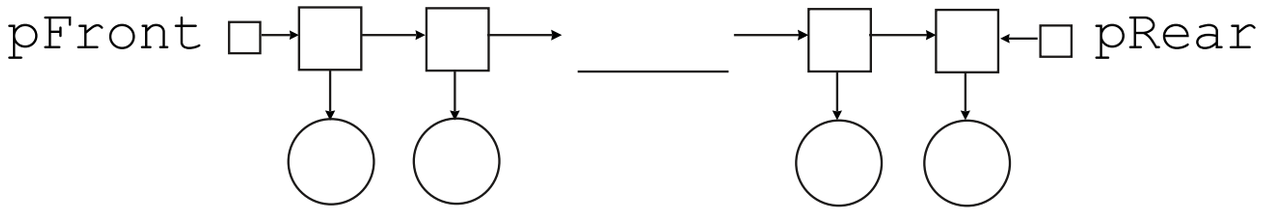
```
void *pv:
stack *pMyStack;
pMyStack = pop(pMyStack, &pv);
cout << ((PERSON *)pv)->pName << endl;
```



Queue

A queue or FIFO ("first in first out") is a list in which the insertions can be perfomed in only one fixed position, called the rear or tail. The deletions can be performed only on another fixed position called the front or head. In implementations the rear is mostly the last and the front the first element of list. In other words, the most recently inserted record will be always on the rear (operation enqueue) and only the record on front can be removed (operation dequeue).

```
struct queue_header
{
      void *pRecord = nullptr;
      queue *pNext = nullptr;
};
struct queue
{
      queue_header *pFront = nullptr;
      queue_header *pRear = nullptr;
 };
```



To avoid long iterations store two pointers: to the front and to the rear. Another solution: implement the queue using a circular list.

In double-ended queue or deque the insertion and removal of records can be performed on the both ends: on the front as well as on the rear.

Methods for speeding up the sequential search (1)

In sequential search we have to iterate the sequence from the beginning to the end and on each step compare the key of current record with the given key. When the keys are identical, the searching stops. If we cannot continue because we have reached the end of sequence, the searching has failed. Example:

```
PERSON *pGroup;
const char *pKey = "John Smith";
int i = 0;
for (;  i < n;  i++)
{
    if (!strcmp(pKey, (pGroup + i)->pName))
    {
          break;
     }
}
if (i == n)
{
     cout << "Failure" << endl;
}
```

Methods for speeding up the sequential search (2)

The worst case is when the record is missing: we need to interate until the end of sequence. If we know that the failures will occur rather often, we may first to sort the sequence:

```
PERSON *pGroup;
const char *pKey = "John Smith";
int i = 0, k;
for (;  i < n;  i++)
{
    if ((k = strcmp(pKey, (pGroup + i)->pName)) <= 0)
   {   // if k < 0, pKey < pName and there is no sense to continue
       // for example we search John from Charles, James, Peter, Winfried
       // if we have already reached Peter we may break off
          break;
    }
}
if (k)
{
     cout << "Failure" << endl;
}
```

Methods for speeding up the sequential search (3)

According to the Pareto principle (known also as 80/20 rule) roughly 80% of consequences come from 20% of causes. For searching it means that roughly 80% of queries are about 20% of records.

Consequently, to speed up the sequential search we may reorder the sequence so that the most queried records (20% of total) are at the beginning. However, it is cumbersome or even impossible to extract them.

In self-organizing linked lists the search is organized as follows:

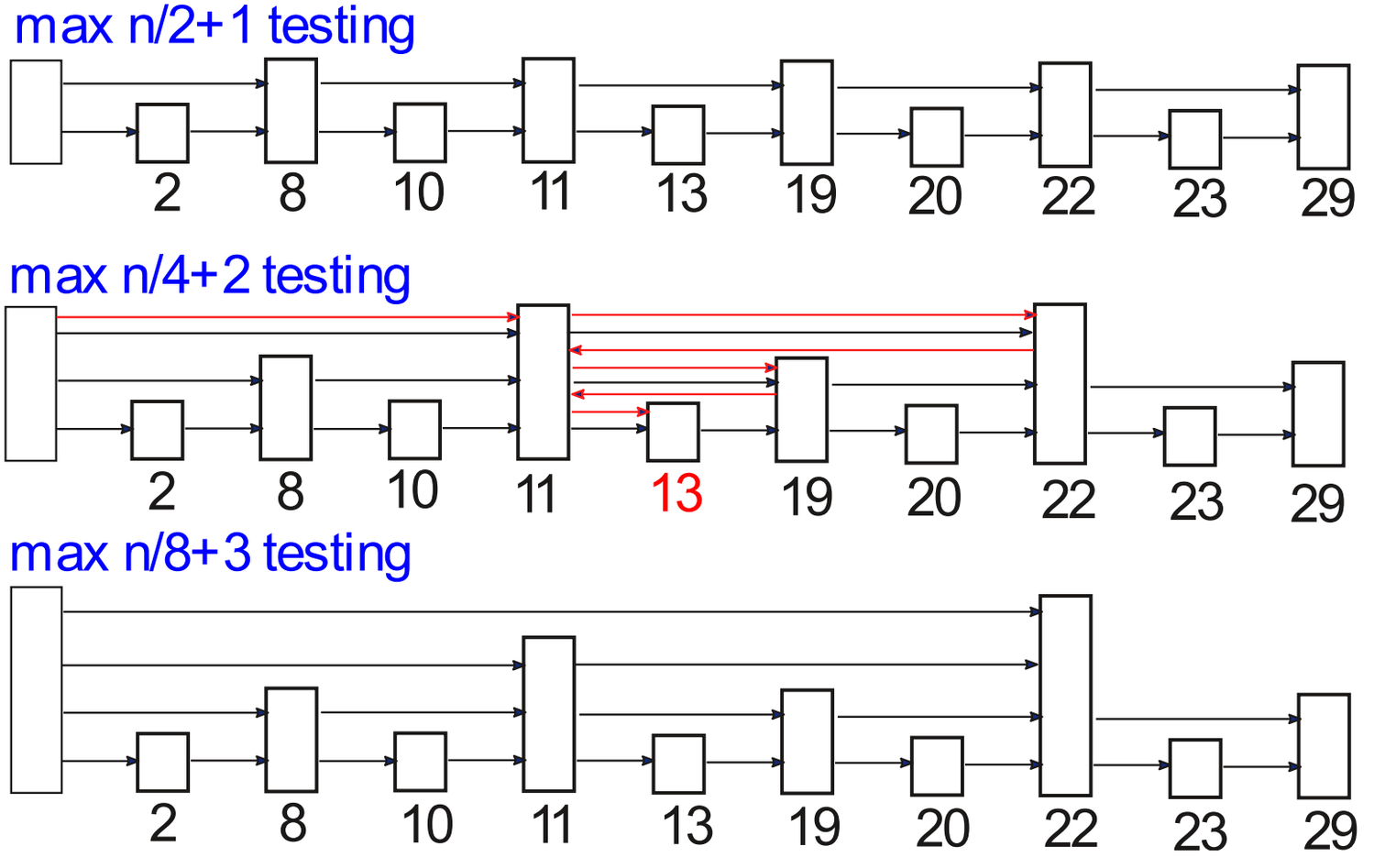
- at the beginning the records are in random order.
- after each successful query the found record is put at the beginning of sequence.

In self-organizing arrays the search is organized as follows:

- at the beginning the records are in random order.
- after each successful query swap the found record with its predecessor.

In both cases after some time the most queried records are located near the beginning.

Skip lists (1)



The records must be sorted. We start moving (red lines) on the upmost level. If we see that we have already jumped over the searched record, return to the previous record, step down to the lower level and continue.

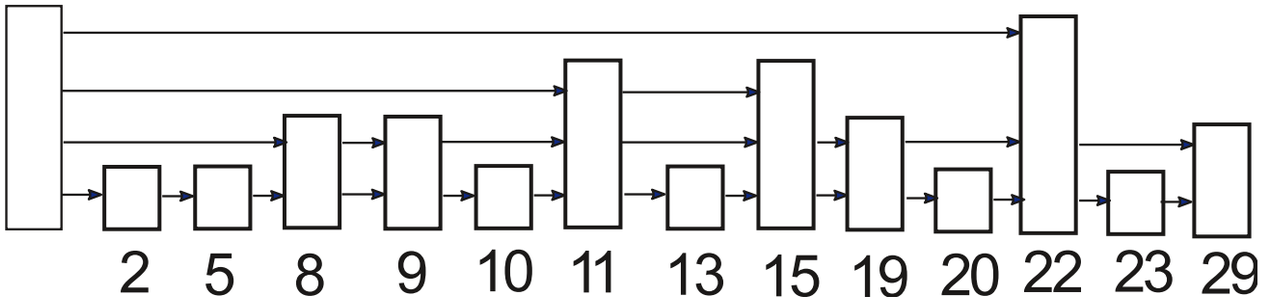
Skip lists (2)

The problem here is that what will happen after inserting some new records. To preserve the original orderly structure we have to do a lot of relocations, actually create a new skip list. It is clear that such a solution is unsuitable.

Note that:
About a half of records are with one pointer.
About a quarter of records are with two pointers.
…….
About $1 / 2^k$ of records are with k pointers.

Let us take k = 4. We need a generator of random numbers that returns:
Integer 1 with probability 0.5.
Integer 2 with probability 0.25.
Integer 3 with probability 0.125.
Integer 4 with probability 0.00625.

The generator output determines how many pointers build into the next new record. In this way a get a skip list that is not absolutely orderly but however, allows faster searching, for example like this one:



Binary search (1)

Let us have a sorted array with length n, for example 15 25 28 30 32 36 37 58 61 68 75. We search record with key 58.

First find the record located in the centre. The index of center is n / 2. In this example 11/ 2 = 5

15 25 28 30 32 36 37 58 61 68 75.

If the record in the centre has the required key, we have finished. If not, take the segment located left (the required key is less than key in the center) or right (the required key is greater than key in the center) from the center and follow in the same way, i.e. find the center, test it and if the result is negative, select the left or right segment:

37 58 61 68 75

37 58

If the length of new segment is zero, the search has failed. For example, if we search 60 we need to select the segment right of 58, but there are no records.

Binary search is recursive but we can also implement it with non-recursive functions. Athough universal in principle, it is practically applicable only to arrays.

Binary search (2)

```
PERSON *BinarySearch(PERSON *pGroup, int n, const char *pKey)
{  // recursive binary search function
   if (!pGroup || !n || !pKey)
   {
       return nullptr;
   }
   PERSON *pCenter = pGroup + n / 2; // pointer to ther center
   int i = strcmp(pCenter->pName, pKey);
   if (!i)
   {
      return pCenter;
   }
   else if (i < 0)
   {  // right segment
      return BinarySearch(pCenter +1, n % 2 ? n / 2 : n / 2 - 1, pKey);
   }
   else
   {  // left segment
      return BinarySearch(pGroup, n / 2, pKey);
    }
}
```

Binary search (3)

```
PERSON *BinarySearch(PERSON *pGroup, int n, const char *pKey)
{  // non-recursive binary search function
   if (!pGroup || !n || !pKey)  {
       return nullptr;
   }
   for (PERSON *p = pGroup;  n > 0; )  {
      PERSON *pCenter = p + n / 2;
      int i = strcmp(pCenter->pName, pKey);
      if (!i)  {
         return pCenter;
      }
      else if (i < 0)  {  // right segment
          p = pCenter + 1;
          n = n % 2 ? n / 2 : n / 2 -1;
       }
       else {  // left segment
           n /= 2;
       }
    }
    return nullptr;
}
```

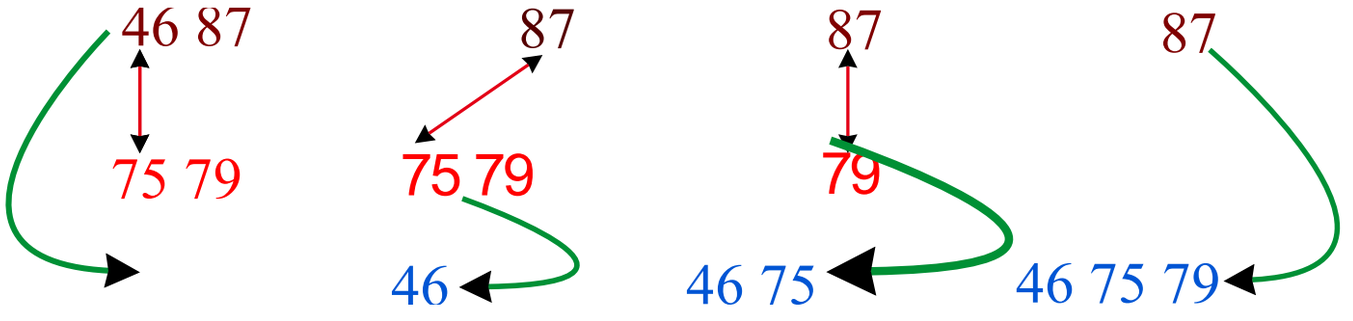
Merge sort (1)

Let us have a linked list of records with keys 87 46 79 75 12 29 64 13 91 95 86 81 21 50.
First create pairs and sort them:

87 46 79 75 12 29 64 13 91 95 86 81 21 50

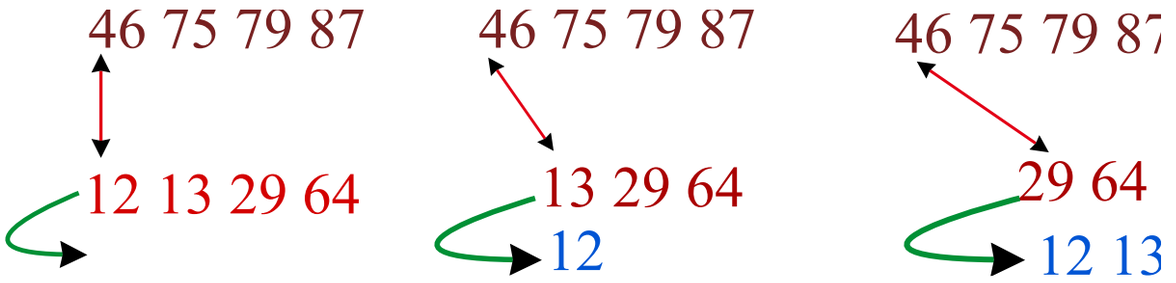
46 87 75 79 12 29 13 64 91 95 81 86 21 50

From sorted pairs merge sorted groups of four records:

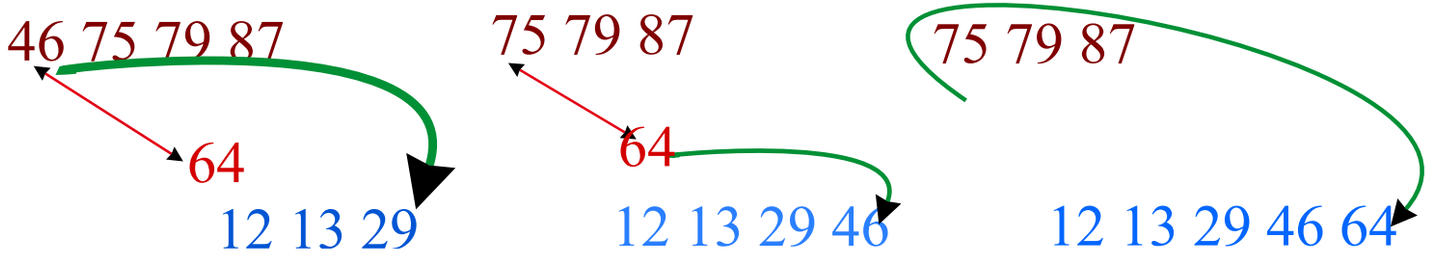


46 75 79 87 12 13 29 64 81 86 91 95 21 50

Next merge sorted groups of eight records:

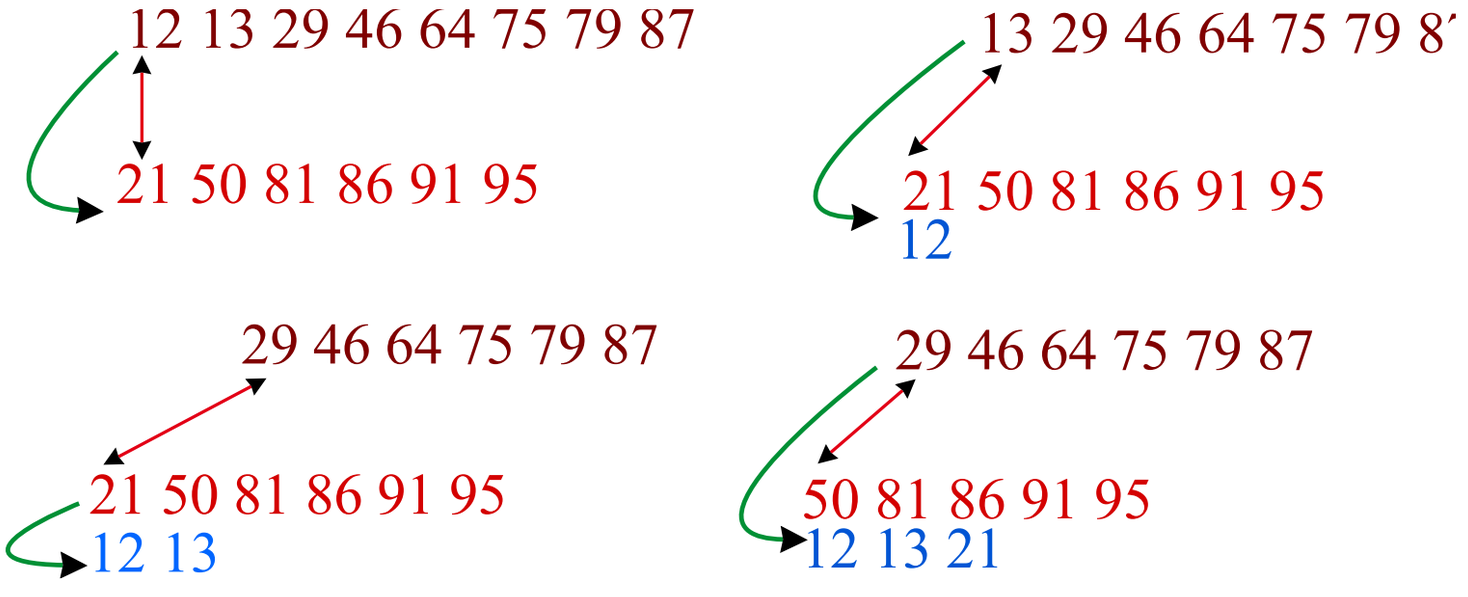


Merge sort (2)

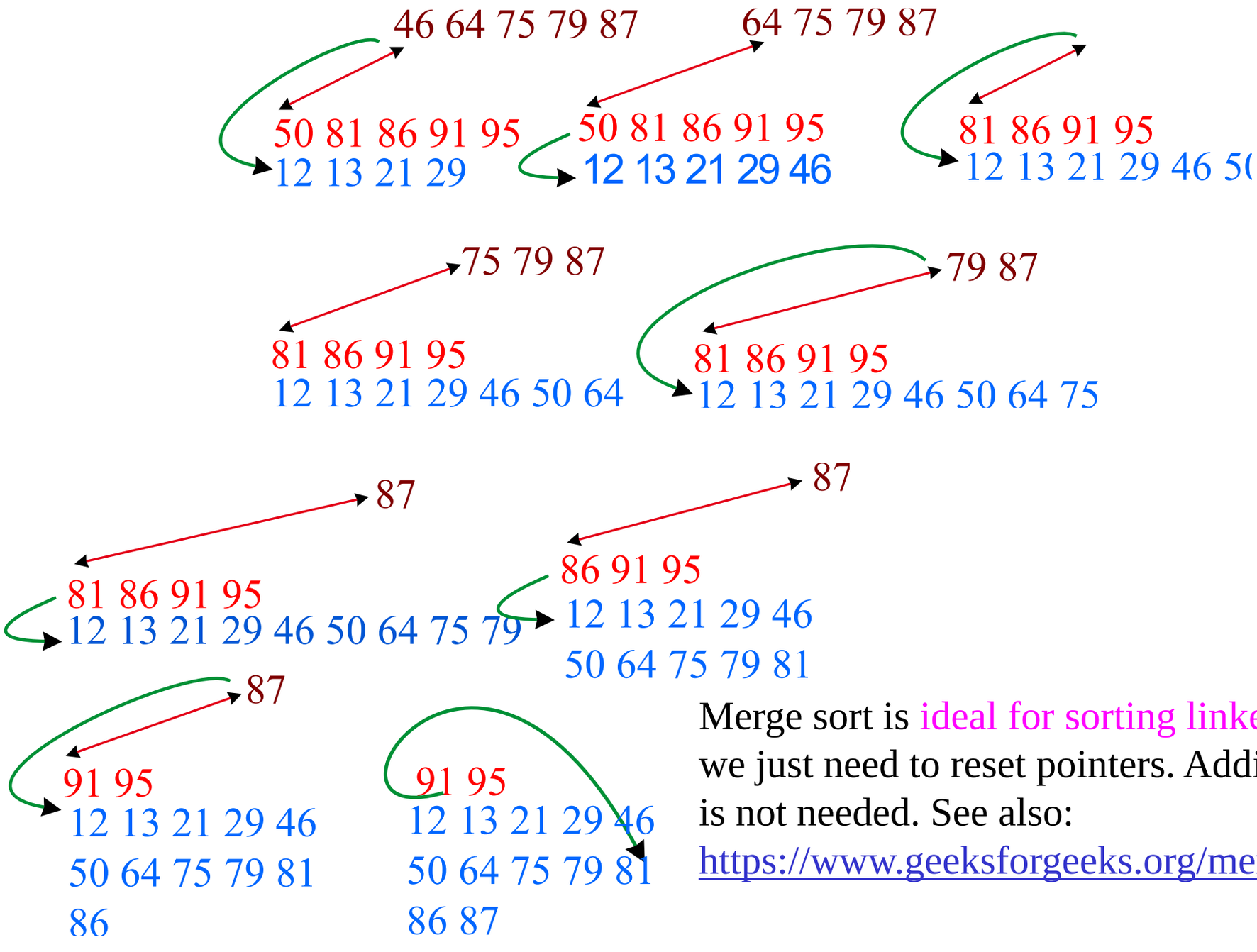


12 13 29 46 64 75 79 87 21 50 81 86 91 95

At last from two sorted halfs merge the complete sorted list:



Merge sort (3)



Merge sort is ideal for sorting linked lists because we just need to reset pointers. Additional memory is not needed. See also:
https://www.geeksforgeeks.org/merge-sort/

C standard functions for searching and sorting

Binary search from a sorted array:

```
void *bsearch (const void *pKey, const void *pArray, size_t nRecords, size_t RecordSize,
                    int (*pCompare) (const void *, const void *));
```

Quick sort for array:

```
void qsort (const void *pArray, size_t nRecords, size_t RecordSize,
                    int (*pCompare) (const void *, const void *));
```

Here:

pKey – pointer to the required key
pArray – pointer to the beginning of array
nRecords – number of members (records) in array
RecordSize – length of one record in bytes
pCompare – pointer to function performing the comparison of keys (will be discussed later)
size_t – actually *unsiged int*, see https://en.cppreference.com/w/c/types/size_t

Example:

```
#include "stdlib.h"
int Compare(const void *pKey, const void *pRecord)
{
     return strcmp((const char *)pKey), ((const PERSON *)pRecord)->pName);
}
PERSON *pJohn = bsearch("John Smith", pGroup, nStudents, sizeof(PERSON), Compare);
```