Lineary data structures

Data Structures and Algorithms

Lineary data structures

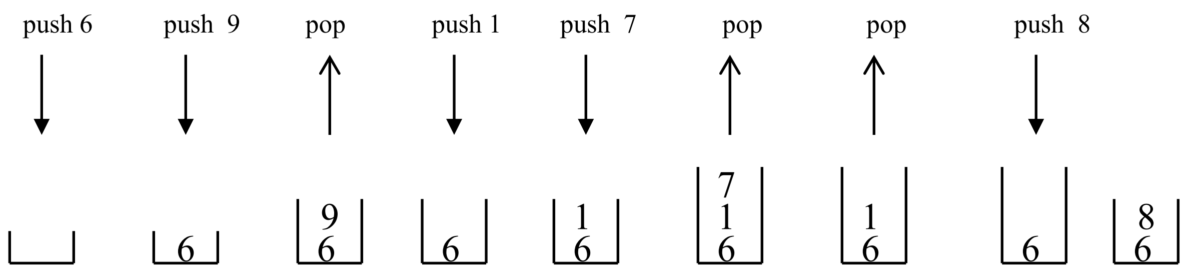
Lineary data structures:

- stacks
- queues
- lists
  - unordered singly-linked list
  - ordered doubly-linked list

Stack

Stack is an abstract data structure, in which an element can be inserted and removed only on one end. Stack is a LIFO structure (*last in, first out*), it means last inserted element will be the first removed element.

Basic operations on a stack:
Init(*stack*) – empties, or preparing the structure to work
Empty(*stack*) – return true if the stack is empty
Full(*stack*) - return true if the stack is full
Push(*el*, *stack*) – push an element on the top of the stack
Pop(*stack*) – pop an element from the top of the stack



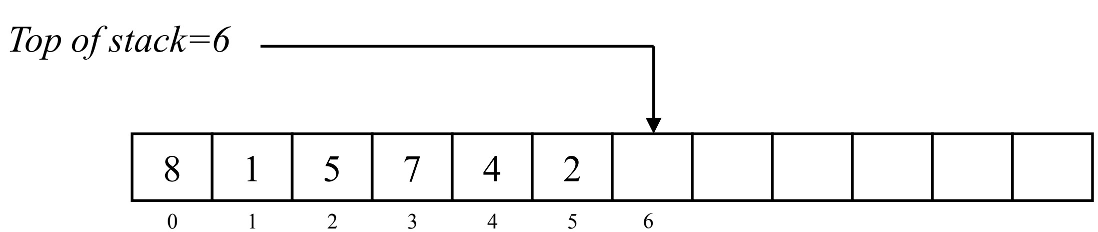
operation sequence on a stack

Stack - realizations

Different representations of a stack in computer (program)

- an array with one organizing index
    - *limited capacity*
    - *better for one-type stack*
- a list
    - *„unlimited" capacity*
    - *different type of element can be used*

Stack realized as an array



Stack (an array)

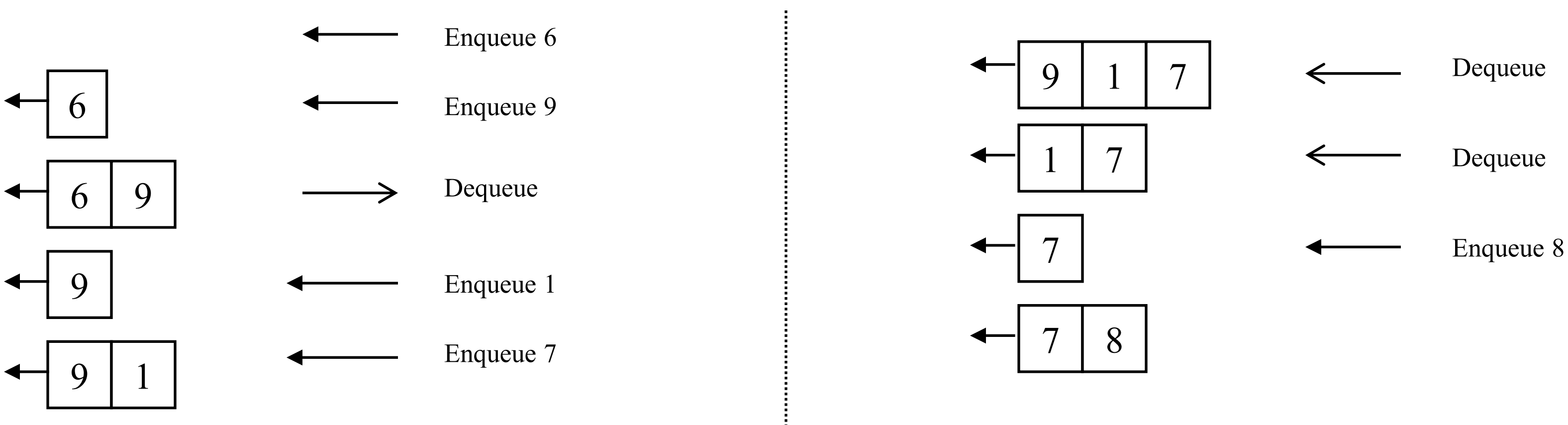
```
typedef struct{
  int *arr;
  int size;
  int top;
} Stack;

void init(Stack &stack, int
    size)
{
  stack.top=0;
  stack.arr=new int[size];
  stack.size=size;
}

bool empty(Stack stack)
{
  return stack.top==0;
}

bool full(Stack stack)
{
  return stack.top==stack.size;
}
```

```
bool push(Stack &stack, int elem)
{
  if(full(stack))
    return false;
  stack.arr[stack.top++]=elem;
  return true;
}

bool pop(Stack &stack, int &elem)
{
  if(empty(stack))
    return false;
  elem=stack.arr[--stack.top];
  return true;
}
```

Queue

is a structure for waiting persons, in which someone can come and stand on the end and someone from the front can go through. Queue is a FIFO structure (*first in, first out*), it means last inserted element will be the last taken element.

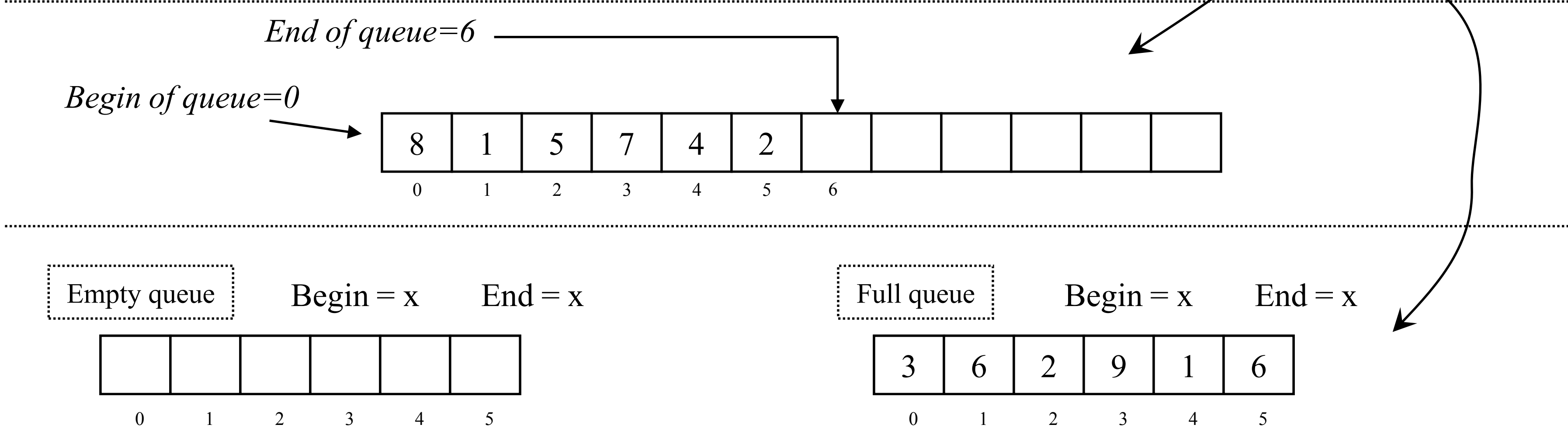
Basic operations on a queue:
Init(*queue*) - empties, or preparing the structure to work
Empty(*queue*) - return true if the queue is empty
Full(*queue*) - return true if the queue is full
Enqueue(*el*, queue) – add an element to the queue
Dequeue(*queue*) – return and delete the first element from the queue



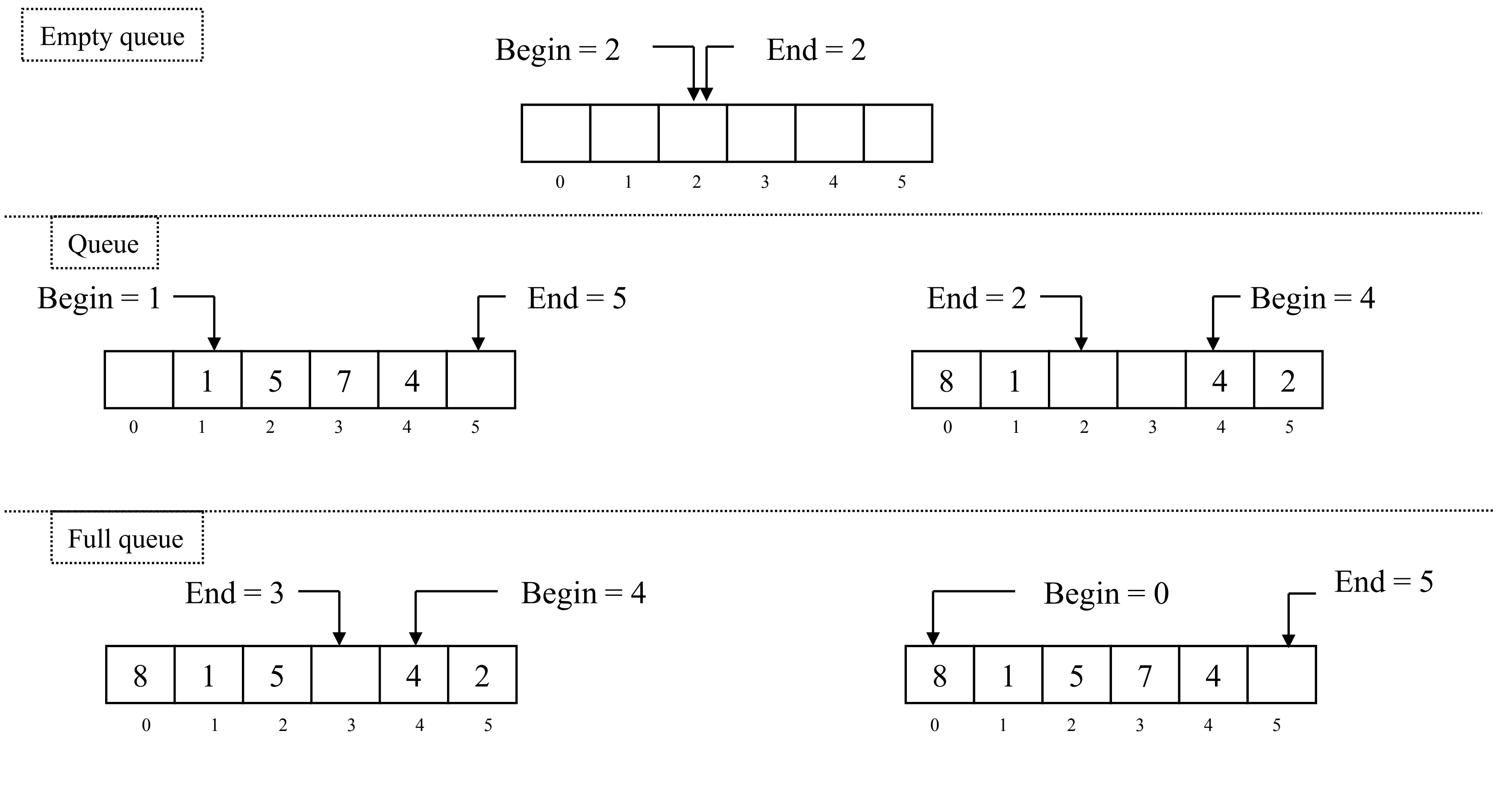
Queue - realizations

Queue representations in computer programs

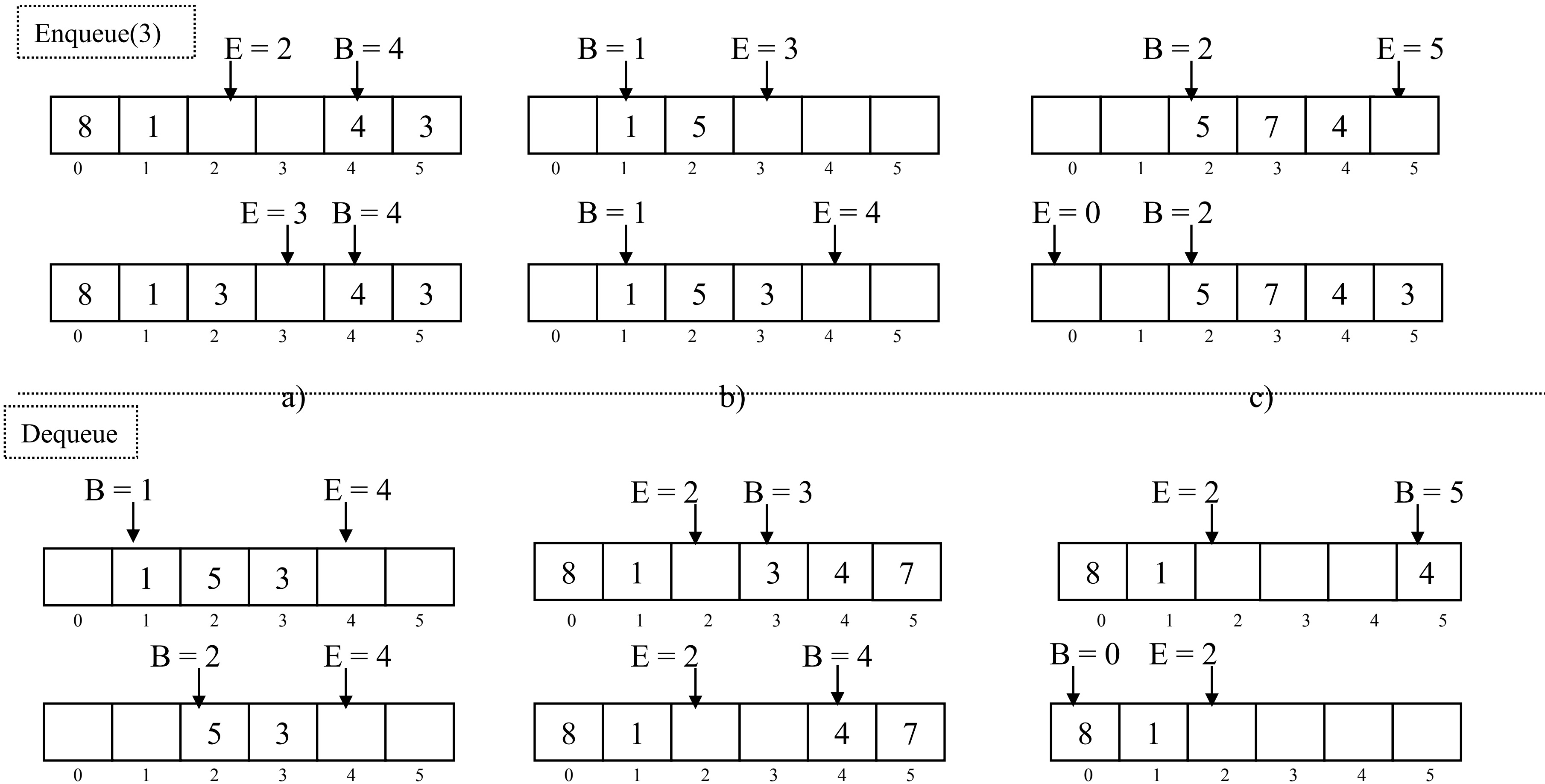
- an array with one organization index
  - similar to stack representation with elements shift
- an array with two organization indexes
  - with „empty” position
  - without „empty” position
- a list



Queue (array with „empty” position)



Queue(cont.) – enqueue, dequeue



Queue (an array)

```
typedef struct
{
  int *arr;
  int size;
  int begin;
  int end;
} Queue;

void init(Queue &queue, int size)
{
  queue.begin=0;
  queue.end=0;
  queue.arr=new int[size+1];
  queue.size=size+1;
}
bool empty(Queue queue)
{
  return queue.begin==queue.end;
}
bool full(Queue queue)
{
  return (queue.begin==0 && queue.end==queue.size-1)
              || (queue.begin==queue.end+1);
}
```

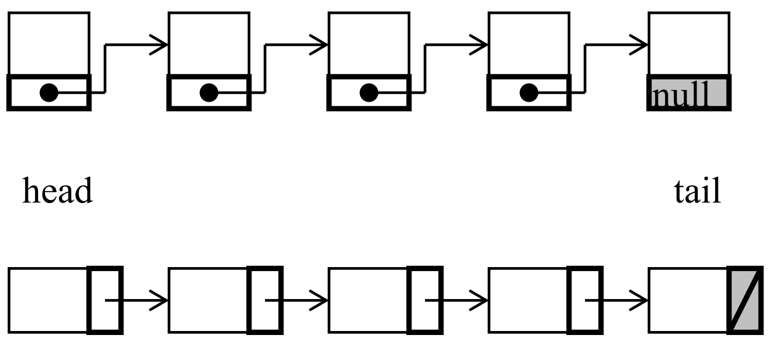
```
bool enqueue(Queue &queue, int elem)
{
  if(full(queue))
    return false;
  queue.arr[queue.end++]=elem;
  if(queue.end>=queue.size)
    queue.end=0;
  return true;
}

bool dequeue(Queue &queue, int &elem)
{
  if(empty(queue))
    return false;
  elem=queue.arr[queue.begin++];
  if(queue.begin>=queue.size)
    queue.begin=0;
  return true;
}
```

Linked List

A ***linked list*** is a data structure in which the objects are arranged in a linear order. The order in a linked list is determined by a reference (pointer) in each object

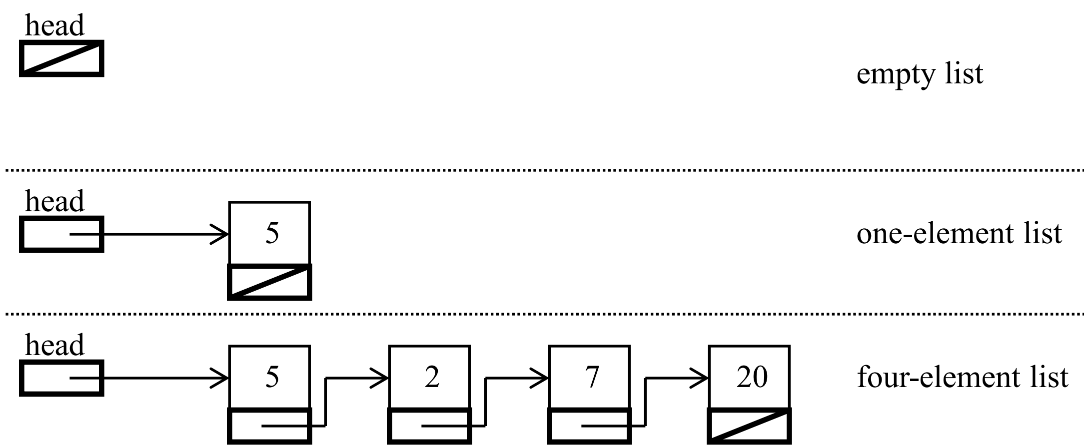
An ***element*** of a linked list is implemented as a record type and have to have minimum two fields: ***a key*** and a reference to a next element. If an element do not have a predecessor, it is called ***a head***. If an element does not have a successor, it is called ***a tail***.

Different graphic representation



Linked List (cont.)

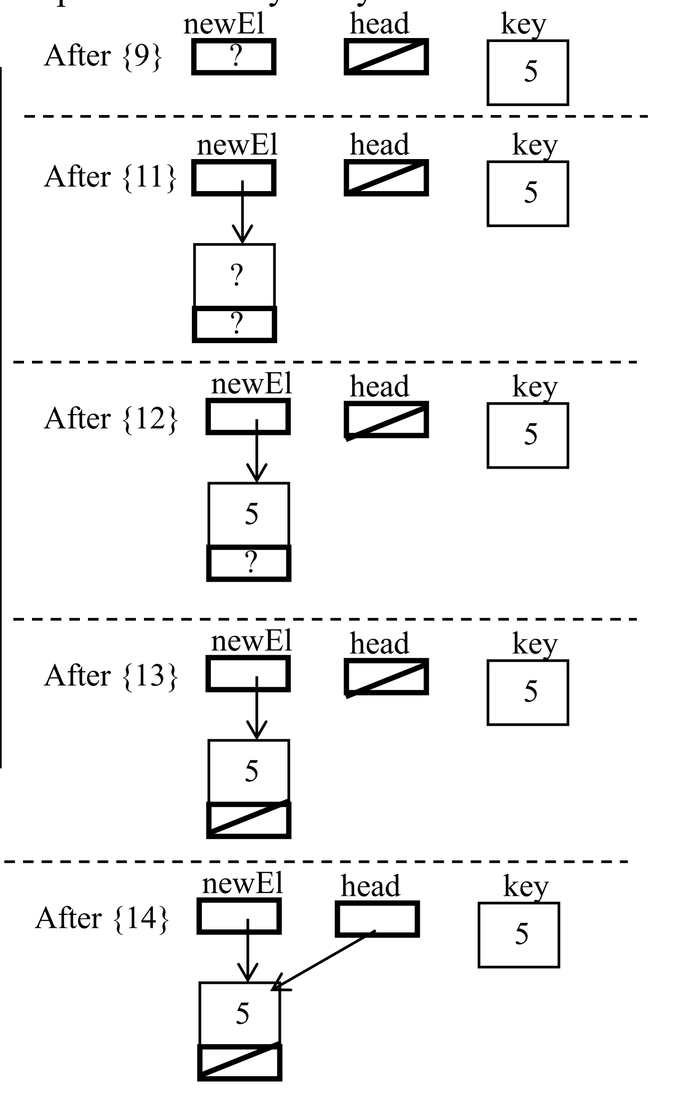
A reference is often an address, under which there is a next element. So we need a reference to the first element of list to have an access to any element. This reference is often stored in a variable called .
If head=*null* then the list is empty.



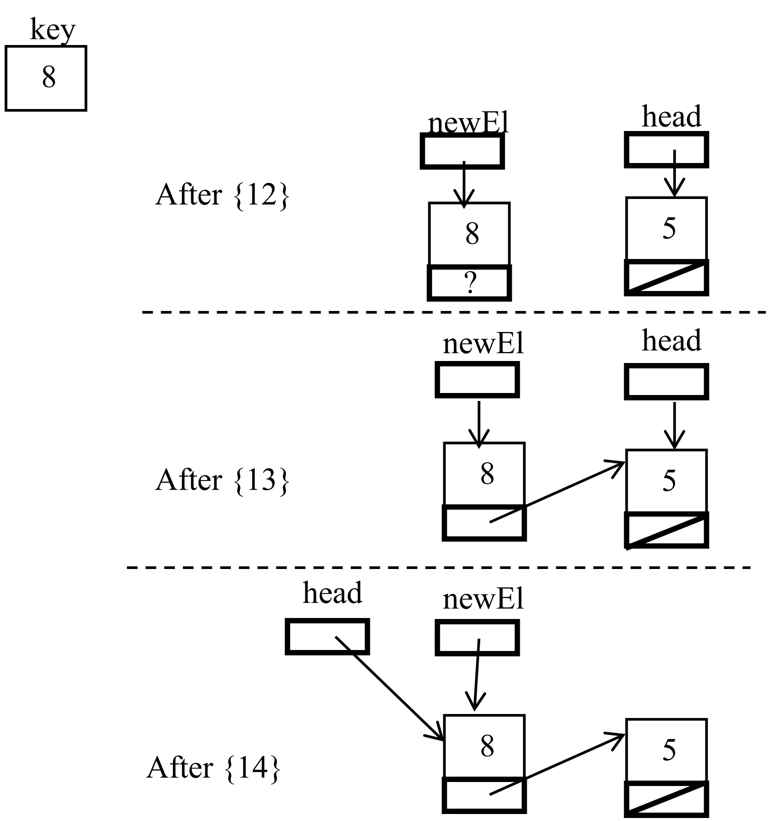
Linked List – insertAsHead

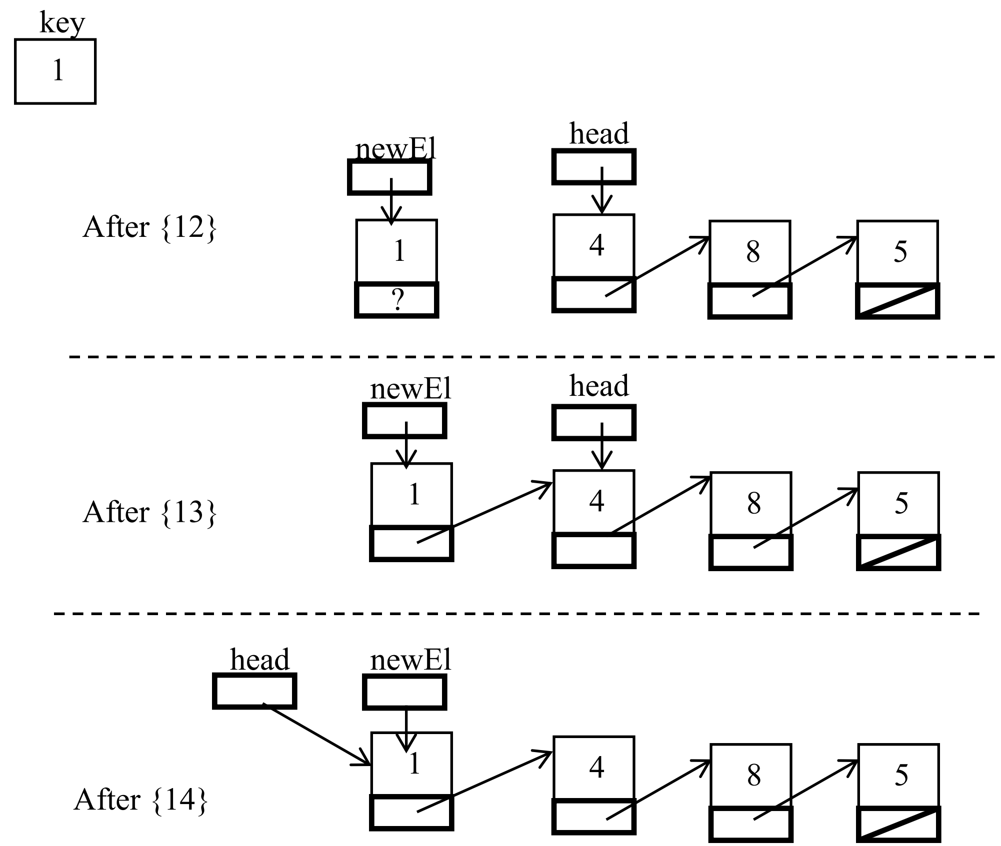
An element of a list can have some additional dates besides a key. But for simplification only a key will be used.

```
1  typedef struct TagElemLL
2  {
3    int key;
4    TagElemLL *next;
5  } ElemLL;
6
7  typedef ElemLL *LinkedList;
8
9  void insertAsHead(LinkedList &head, int key)
10 {
11   ElemLL *newEl=new ElemLL;
12   newEl->key=key;
13   newEl->next=head;
14   head=newEl;
15 }
```



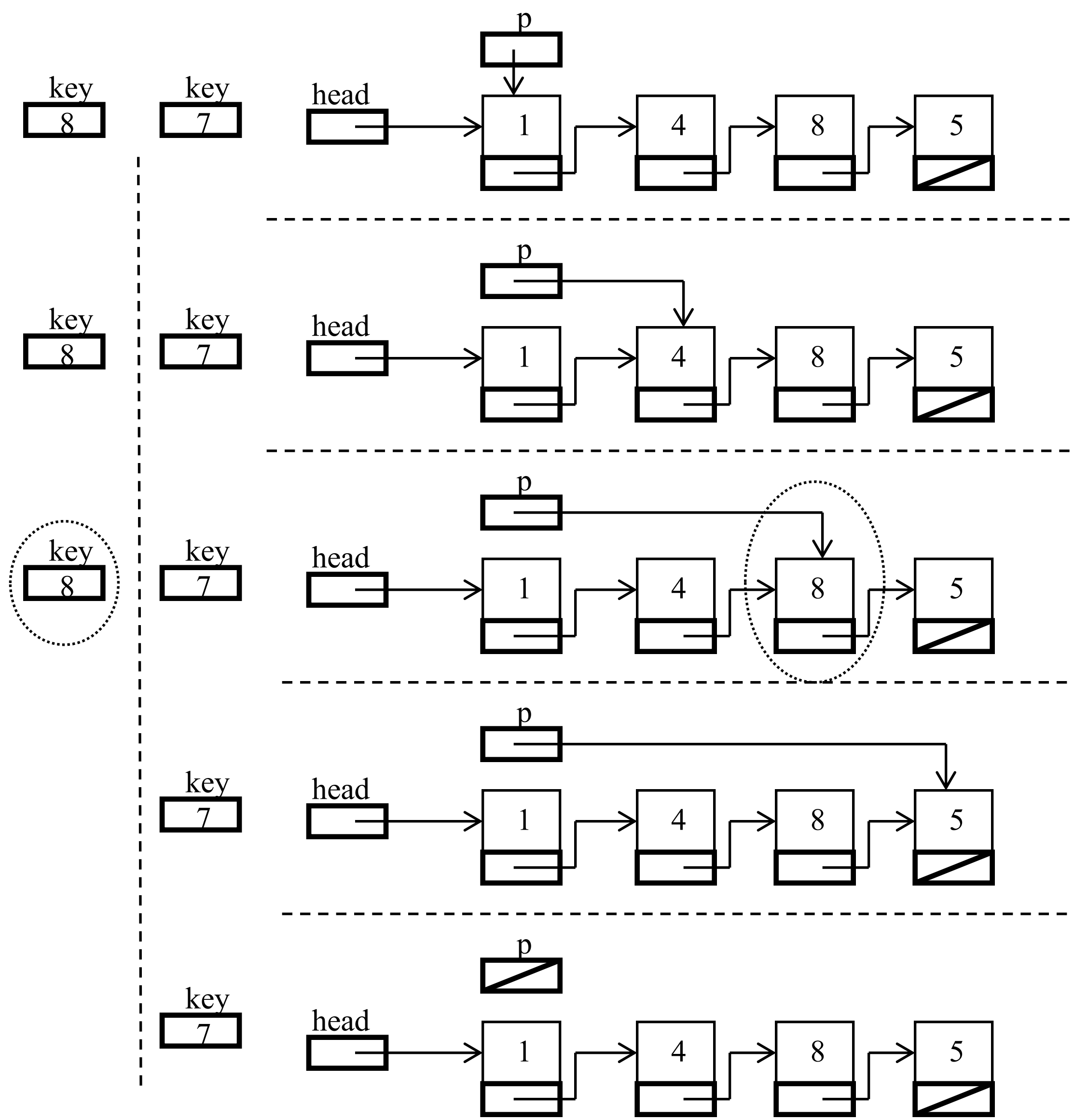
Linked List – insertAsHead (cont.)





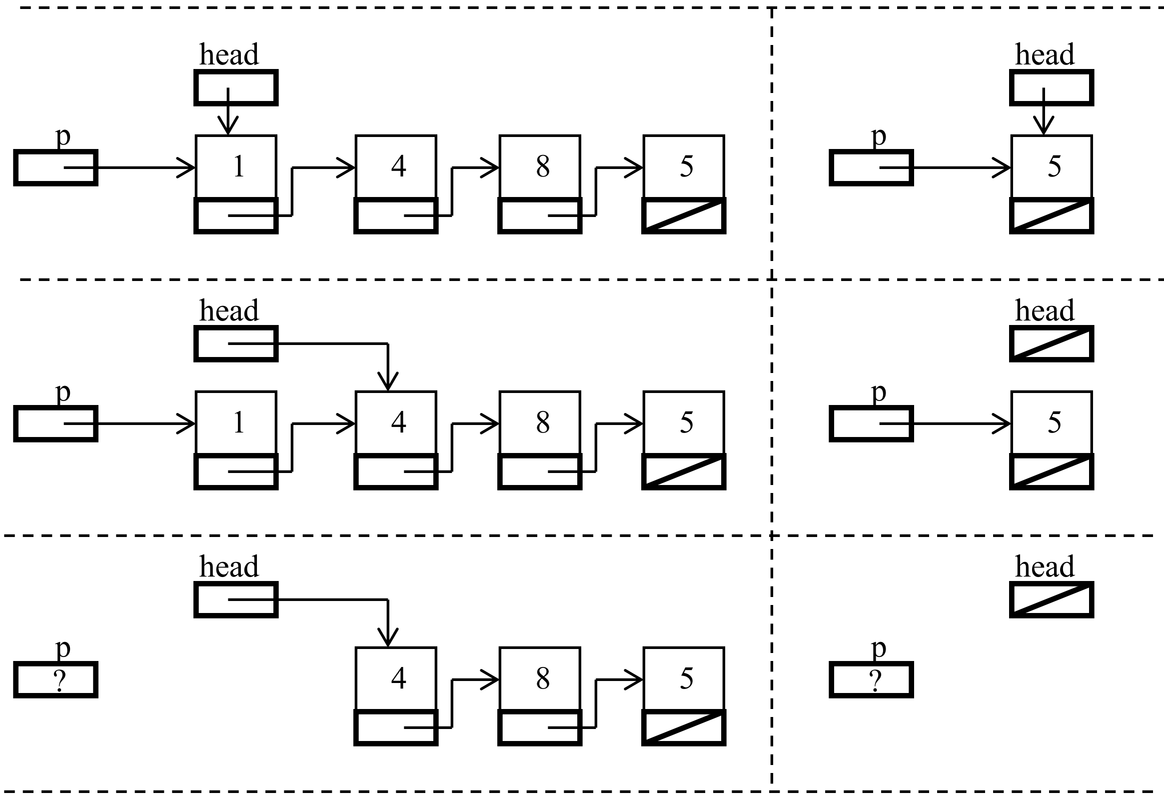
Linked List – findElem

```
ElemLL *findElem(
     LinkedList head, int key)
{
  ElemLL *p=head;
  while(p!=null && p->key!=key)
    p=p->next;
  return p;
}
```

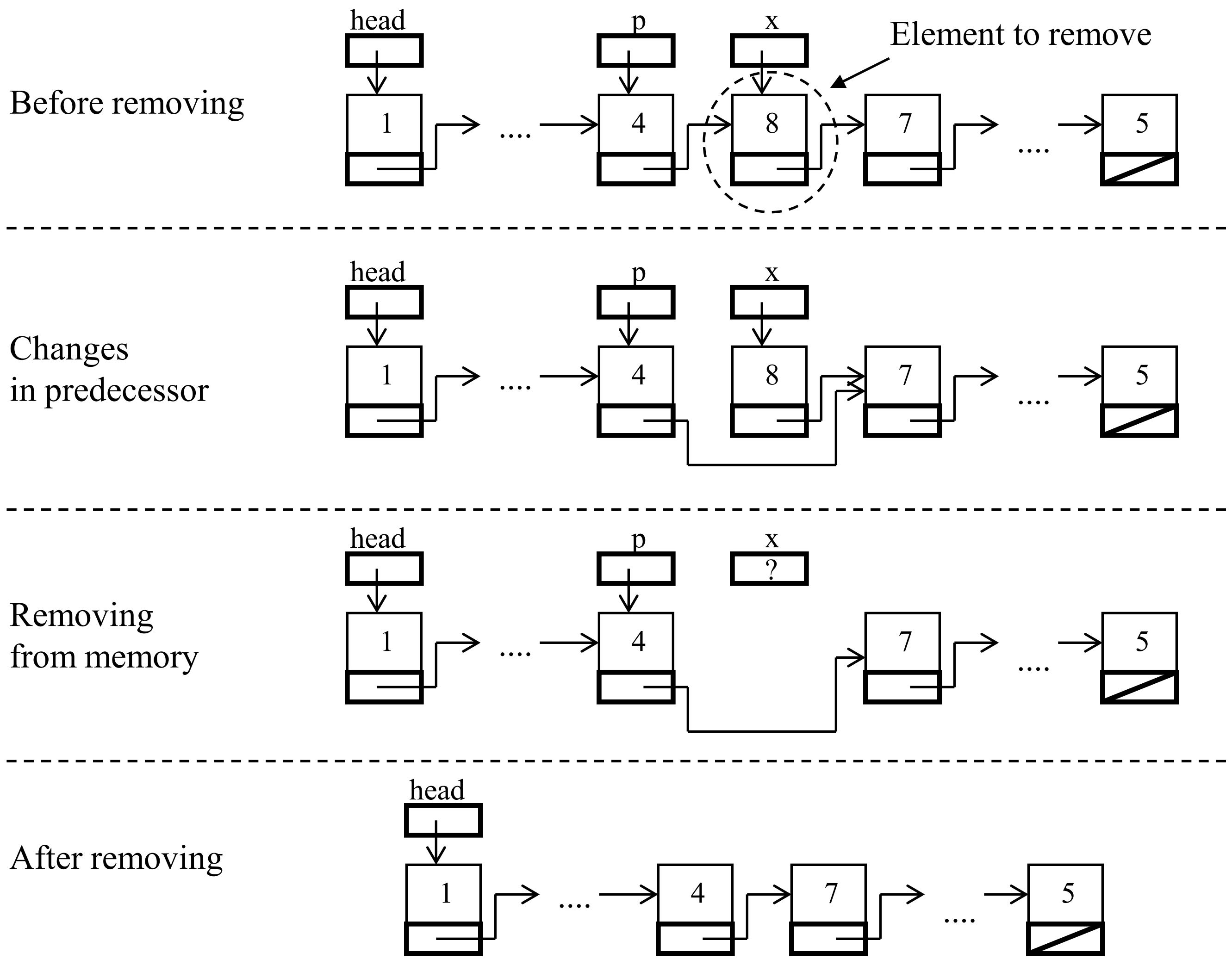


Linked list – removeHead

```
void removeHead(LinkedList &head)
{
  if(head!=null)
  {
    ElemLL *p=head;
    head=head->next;
    delete p;
  }
}
```



Linked list - removeElem

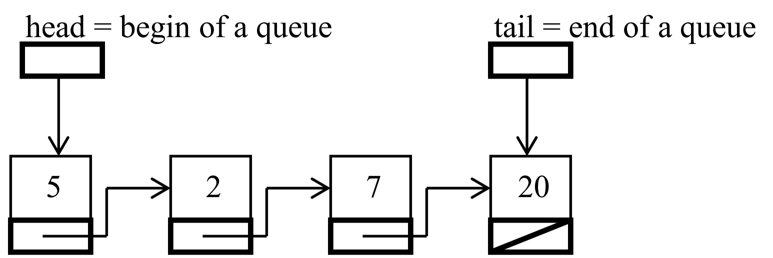


Linked list - removeElem

```
void removeElem(LinkedList &head, int key)
{
  if(head!=null)
    if(head->key==key)
      removeHead(head);
    else
    {
      ElemLL *p=head,*x;
      while(p->next!=null && p->next->key!=key)
        p=p->next;
      if(p->next->key==key) // WRONG !!!
      {
        x=p->next;
        p->next=x->next;
        delete x;
      }
    }
}
```

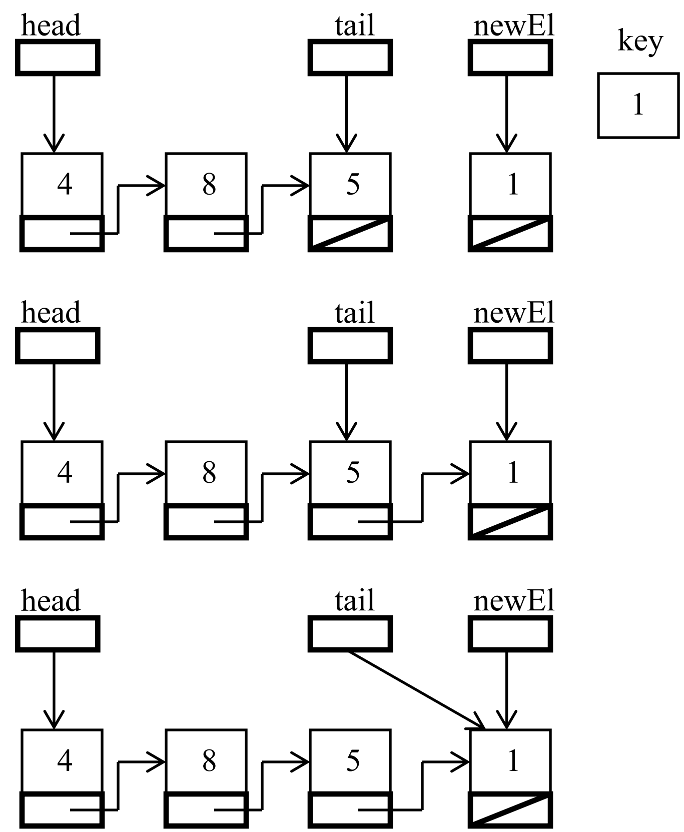
Linked list as a stack or a queue

- **Linked list with a head** (one organising reference) is suitable for **stack** implementation. Pushing on stack is realised by inserting as a head and popping an element – as removing a head. Such a stack is of unlimited capacity.
- **Linked list** can be used also for queue implementation. but because of optimisation besides of **head** we need a pointer to a **tail**.



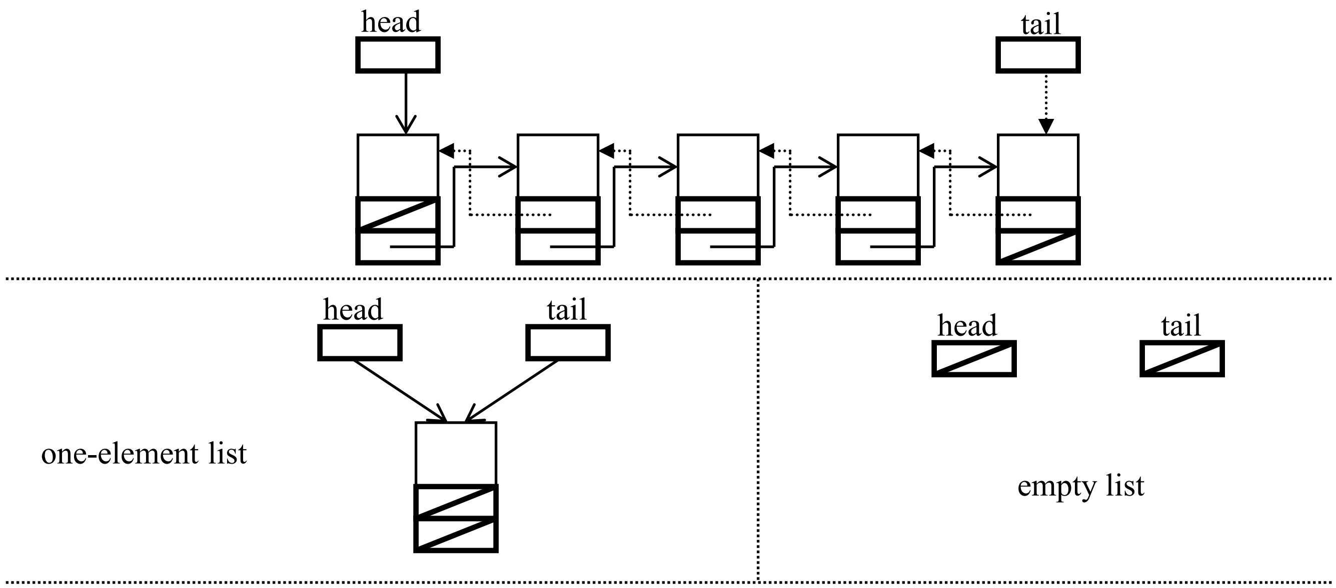
Linked list – insertAsTail

```
typedef struct
{
  ElemLL *head,*tail;
} LinkedList;


void insertAsTail(LinkedList &list, int key)
{
  ElemLL *newEl=new ElemLL;
  newEl->key=key;
  newEl->next=null;
  if(list.tail!=null)
    list.tail->next=newEl;
  else
    list.head=newEl;
  list.tail=newEl;
}
```



Double linked list

An element of **doubly-linked list (two-way linked list)** has two pointers. The first is an address for successor, the second – for predecessor. As **singly-linked list** (**one-way linked list**), double linked list can have one or two one organising pointers.



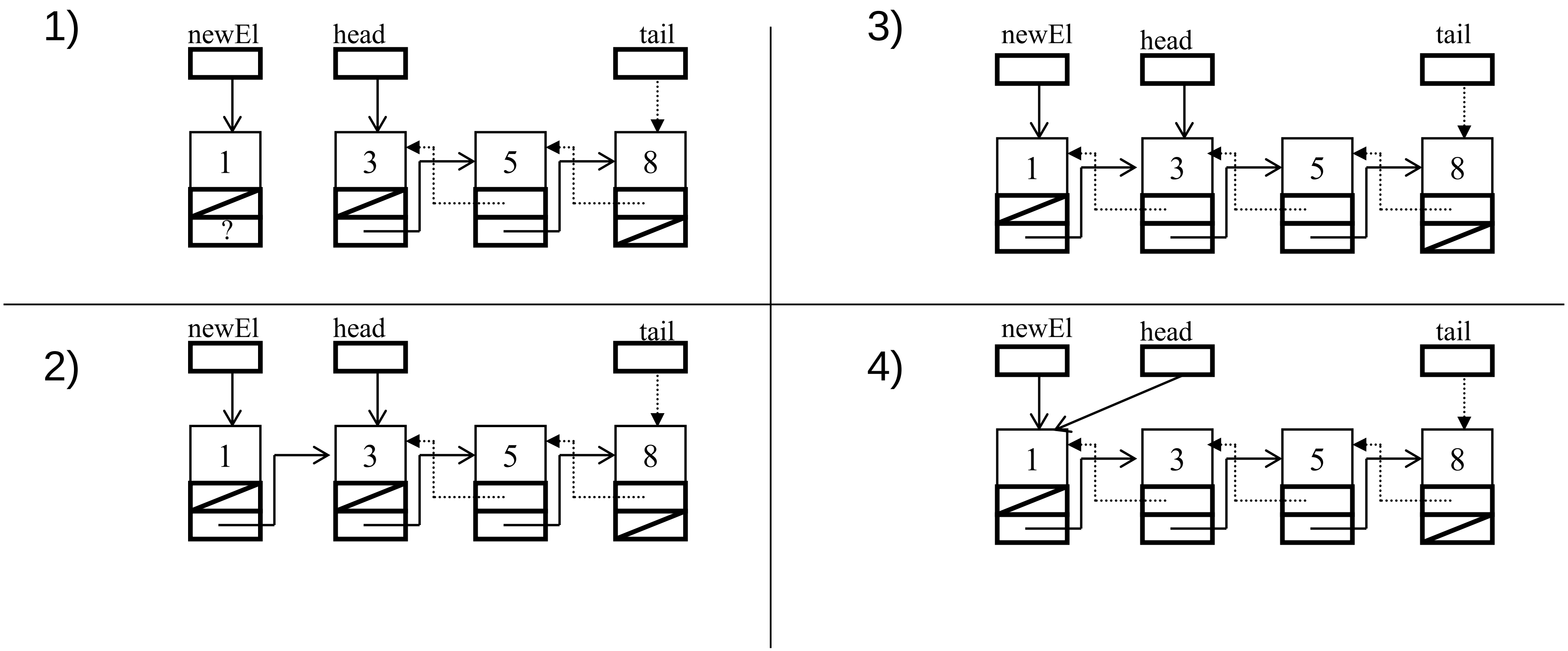
Let's consider a double linked list **ordered by a key**. The searching for element with specific key is similar as for single linked list. But the inserting and removing procedure are different.

Double linked list - insertElem

During inserting a new element into sorted list we have to consider 4 situation: Inserting:

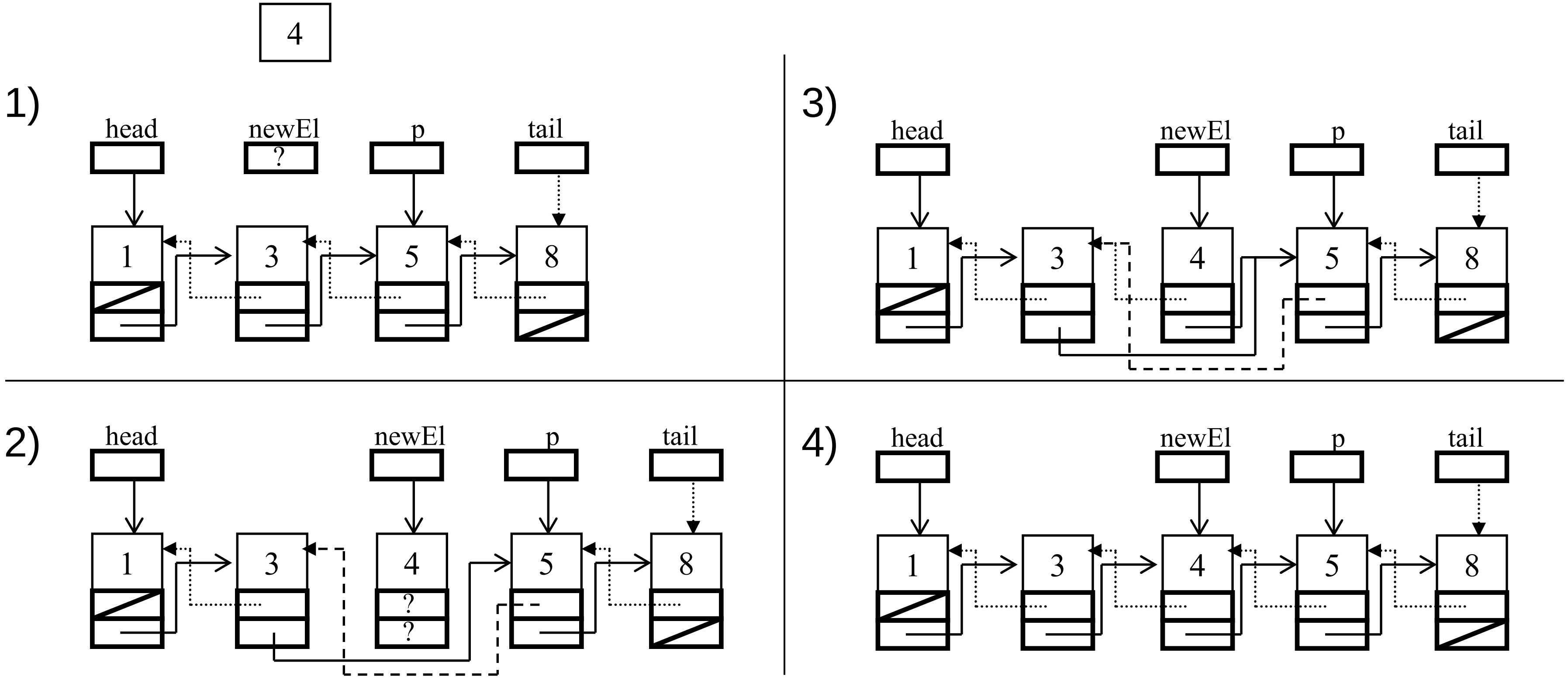
- into an empty list
- as a head
- in the middle of the list (after a head and before a tail)
- as a tail

as a head



Double linked list – insertElem …

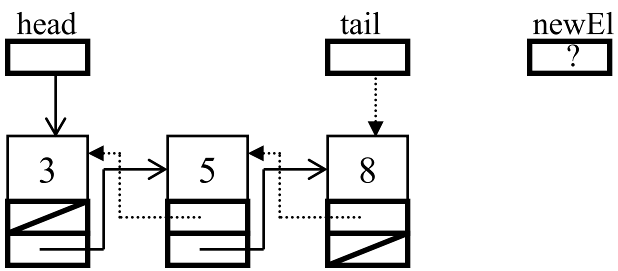
in the middle



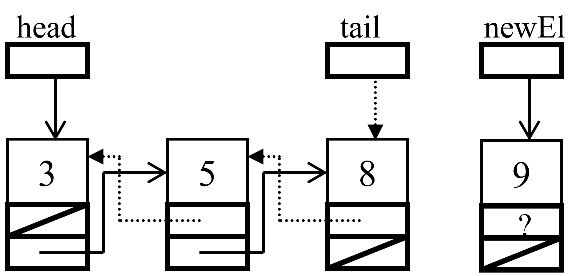
Double linked list – insertElem …

as a tail

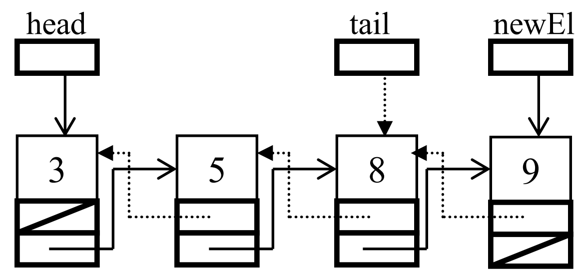
1)



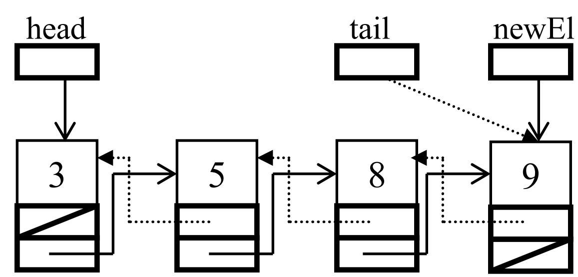
2)



3)



4)



Double linked list – insertElem …

```
typedef struct TagElemLL
{
  int key;
  TagElemLL *next,*prev;
} ElemLL;

typedef struct
{
  ElemLL *head,*tail;
} DoubledLinkedList;
```

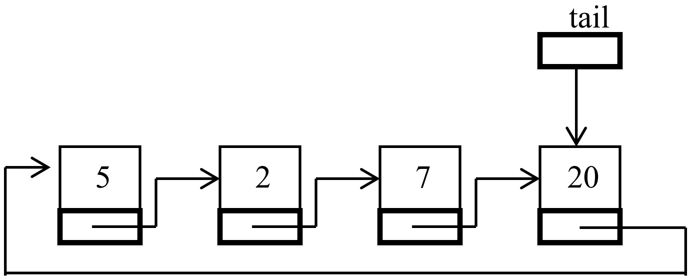
```
void insert(DoubledLinkedList list, int key)
{
  ElemLL *newEl=new ElemLL;
  newEl->key=key;
  ElemLL *p=list.head;
  while(p!=null && p->key<key)
    p=p->next;
  if(p==null)
  {
    newEl->next=null;
    newEl->prev=list.tail;
    if(list.tail!=null)
      list.tail->next=newEl;
    else
      list.head=newEl;
    list.tail=newEl;
  }
  else
  {
    newEl->next=p;
    newEl->prev=p->prev;
    p->prev=newEl;
    if(newEl->prev==null)
      list.head=newEl;
    else
      newEl->prev->next=newEl;
  }
}
```

List - operation

- Basic operations:
  - insert as head
  - insert as tail
  - insert in order (for ordered list)
  - remove head
  - remove tail
  - remove chosen
  - show/compute something for all
  - find
  - count
  - remove all
- Other operation:
  - merge lists
  - reverse list
  - copy list
  - …

List category

- List link:
  - singly-linked – one-way linked
  - doubly-linked – two-way linked
- List order:
  - ordered
  - unordered
- List inner organisation:
  - with head or tail
  - with head and tail
- List end:
  - ordinary-linked
  - circularly-linked
- Specific lists:
  - with sentinel
  - cycled on last element



Advances, disadvances

- list vs arrar
  - list:
    - dynamic(+)
    - unlimited(+)
    - sequential access(-)
    - extra storage(-),
  - array:
    - static(-)
    - limited(-)
    - random access(+)