Doubly linkek lists – Special lists

Basics of Programming 1

G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

26 November, 2025

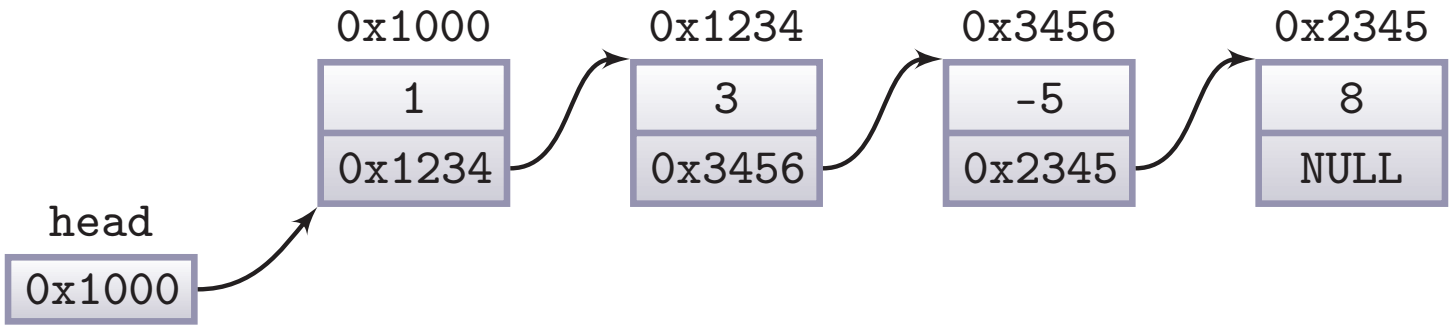
Content

Chapter 1

Singly linked lists

Linked list

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

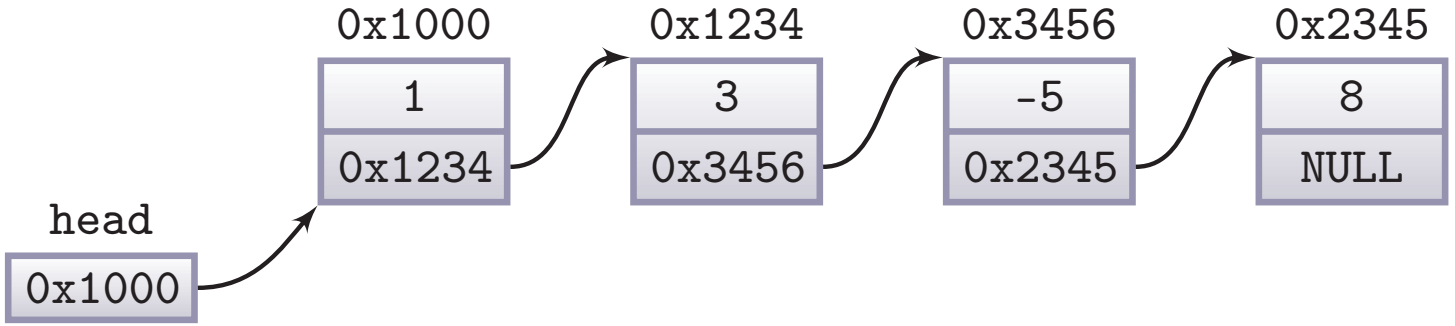


- List of `listelem` type variables
- Memory is allocated dynamically, separately for each element
- Elements do not form a continuous block in memory
- Each element contains the address of the next element
- The first element is defined by the `head` pointer
- The last element points to nowhere (`NULL`)

Linked list

- Empty list

- List is a self-referencing (recursive) data structure. Each element points to a list.



List or array

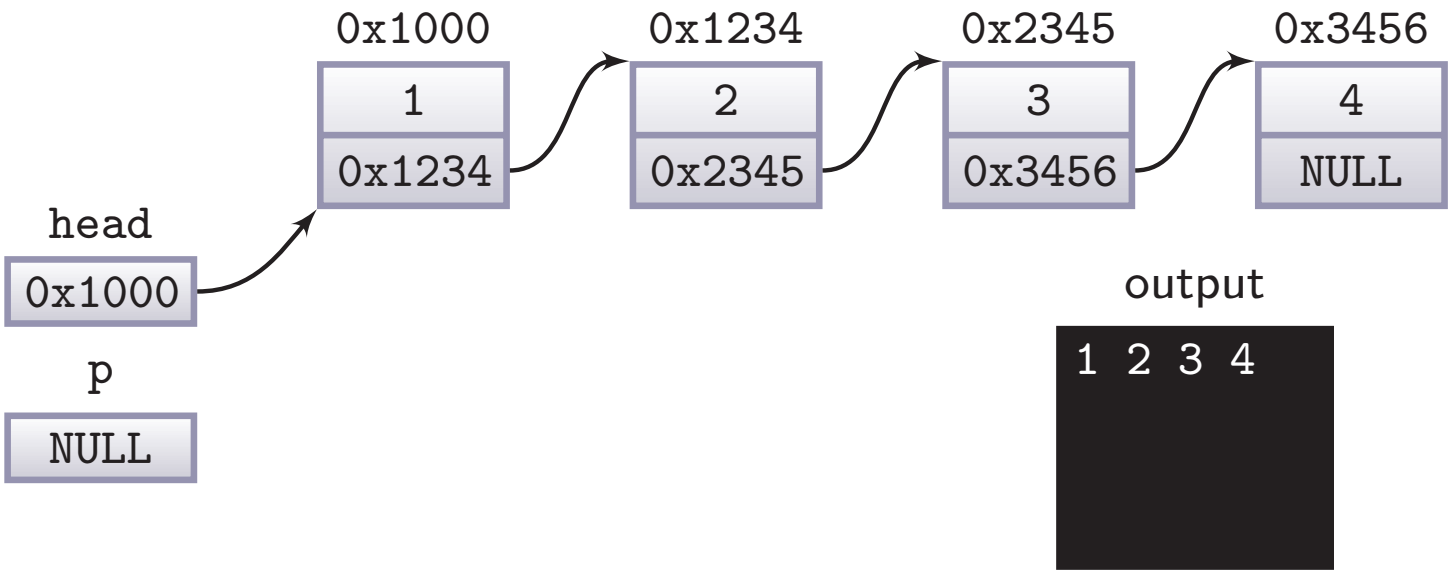
- The array
  - occupies as much memory, as needed for storing the data
  - needs a continuous block of memory
  - any element can be accessed directly (immediately), by indexing
  - inserting a new data involves a lot of copying
- The list
  - elements store the address of the next element, this may need a lot of memory
  - can make use of gaps in the fragmented memory
  - only the next element can be accessed immediately
  - inserting a new element involves only a little work

Traversing a list

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- For traversing we need an auxiliary pointer (p), that will run along the list.

```
1  listelem *p = head;
2  while (p != NULL)
3  {
4    printf("%d ", p->data); /* p->data : (*p).data */
5    p = p->next;            /* arrow operator */
6  }
```



Passing a list to a function

- As a list is determined by its starting address, we only need to pass the startig address for the function

```c
void traverse(listelem *head) {
  listelem *p = head;
  while (p != NULL)
  {
    printf("%d ", p->data);
    p = p->next;
  }
}
```

link

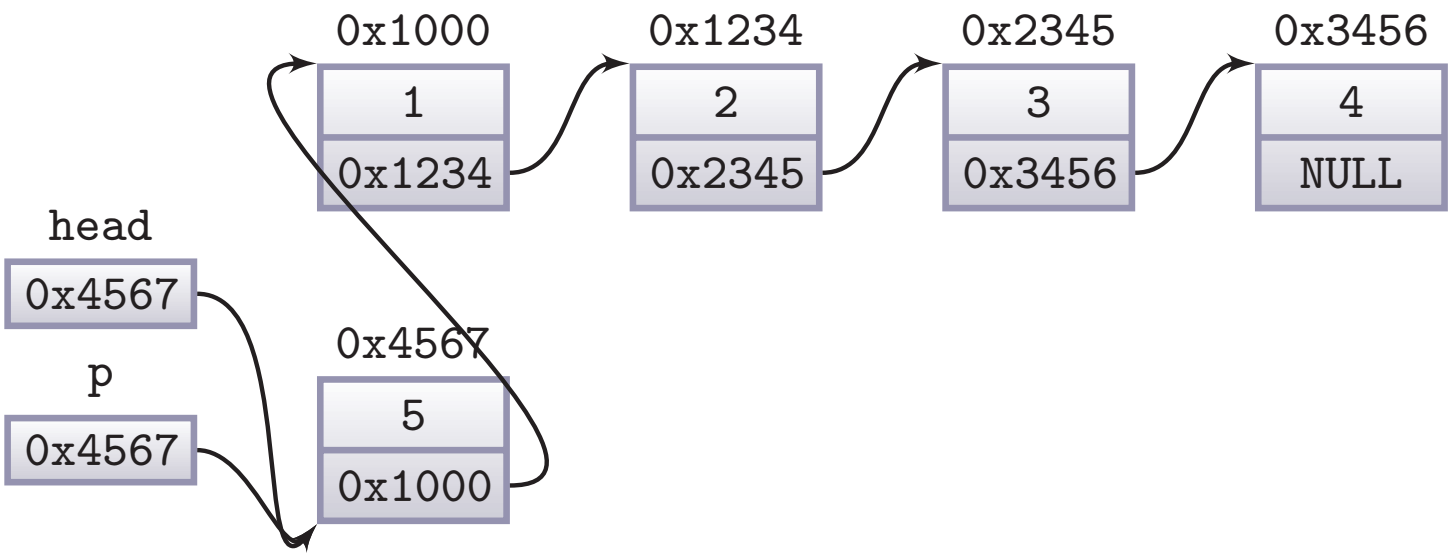
- the same with `for` loop

```c
void traverse(listelem *head) {
  listelem *p;
  for (p = head; p != NULL; p = p->next)
    printf("%d ", p->data);
}
```

Inserting element to the front of the list

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  p = (listelem*)malloc(sizeof(listelem));
2  p->data = 5;
3  p->next = head;
4  head = p;
```



Inserting element to the front of the list, with a function

- As the starting address is changed when inserting, we have to return it (pass it back)

```
1  listelem *push_front(listelem *head, int d)
2  {
3    listelem *p = (listelem*)malloc(sizeof(listelem));
4    p->data = d;
5    p->next = head;
6    head = p;
7    return head;
8  }
```

link

- Usage of function

```
1  listelem *head = NULL;          /* empty list */
2  head = push_front(head, 2);     /* head is changed! */
3  head = push_front(head, 4);
```

Inserting element to the front of the list, with a function

- Another option is to pass the starting address by its address

```
1  void push_front(listelem **head, int d)
2  {
3    listelem *p = (listelem*)malloc(sizeof(listelem));
4    p->data = d;
5    p->next = *head;
6    *head = p; /* *head is changes, this is not lost */
7  }
```

link

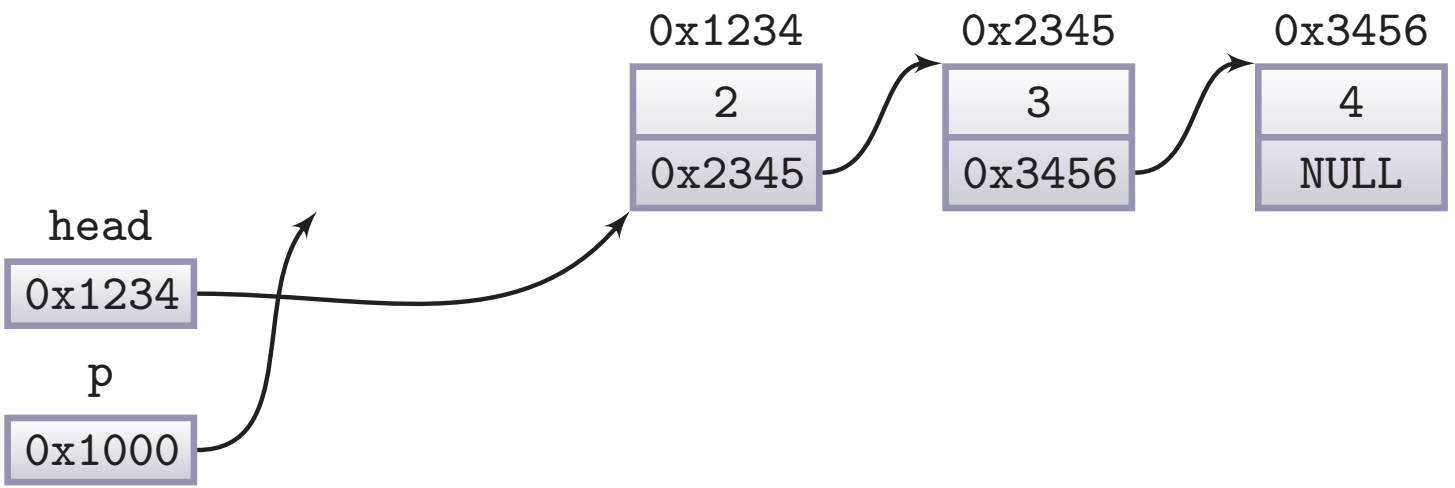
- In this case the usage of the function is:

```
1  listelem *head = NULL;   /* empty list */
2  push_front(&head, 2);    /* calling with address */
3  push_front(&head, 4);
```

Deleting element from the front of the list

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  p = head;
2  head = head->next;
3  free(p);
```



Deleting element from front of the list with a function

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  listelem *pop_front(listelem *head)
2  {
3    if (head != NULL) /* not empty */
4    {
5      listelem *p = head;
6      head = head->next;
7      free(p);
8    }
9    return head;
10 }
```

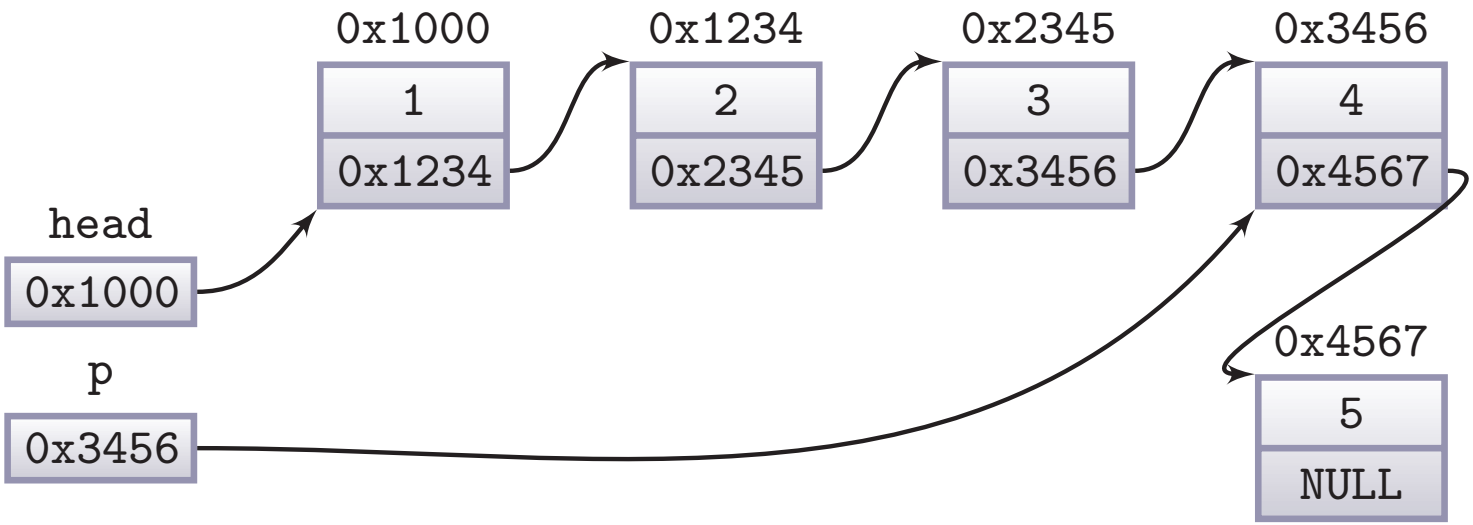
link

- An empty list must be handled separately
- Of course we could use the solution when calling the functionwith the address of `head`

Inserting element to the end of the list

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```
1  for (p = head; p->next != NULL; p = p->next);
2  p->next = (listelem*)malloc(sizeof(listelem));
3  p->next->data = 5;
4  p->next->next = NULL;
```



- If the list is empty, checking `p->next != NULL` is not possible, this case must be managed separately!

Inserting element to the end of the list with a function

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
listelem *push_back(listelem *head, int d)
{
  listelem *p;

  if (head == NULL)   /* empty list should be
            managed separately */
    return push_front(head, d);

  for (p = head; p->next != NULL; p = p->next);
  p->next = (listelem*)malloc(sizeof(listelem));
  p->next->data = d;
  p->next->next = NULL;
  return head;
}
```

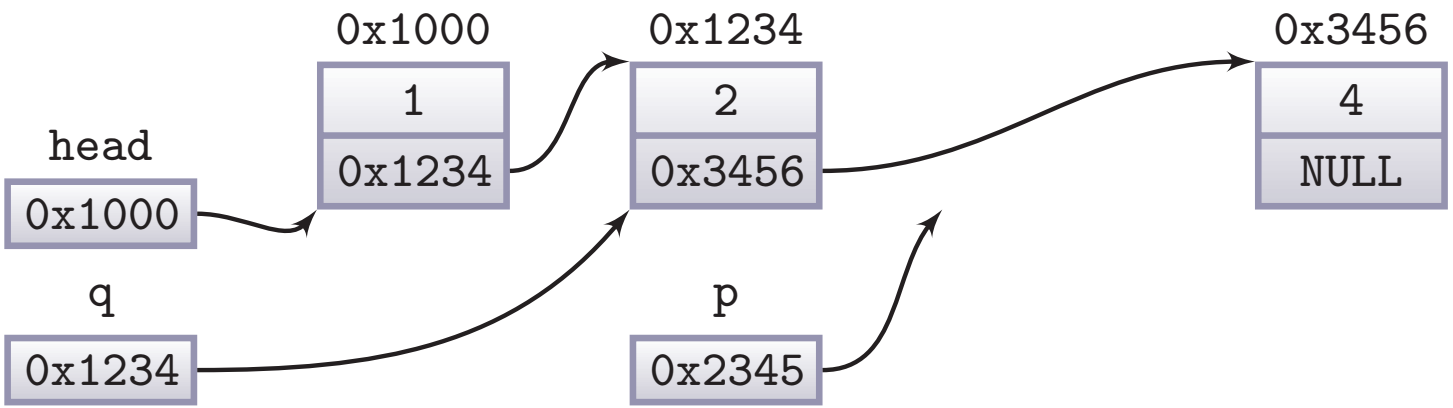
link

```c
listelem *head = NULL;
head = push_back(head, 2);
```

Deleting a given element from list

- Deleting the `data = 3` element

```
1  q = head; p = head->next;
2  while (p != NULL && p->data != data) {
3    q = p; p = p->next;
4  }
5  if (p != NULL) { /* now we have it */
6    q->next = p->next;
7    free(p);
8  }
```



- If the list is empty, or we have to delete the first element, this does not work

Deleting a given element from list


```
1  listelem *delete_elem(listelem *head, int d)
2  {
3    listelem *p = head;
4
5    if (head == NULL) return head;
6
7    if (head->data == d) return pop_front(head);
8
9    while (p->next != NULL && p->next->data != d)
10     p = p->next;
11   if (p->next != NULL)
12   {
13     listelem *q = p->next;
14     p->next = q->next;
15     free(q);
16   }
17   return head;
18 }
```

link

Deleting an entire list


```
1  void dispose_list(listelem *head)
2  {
3    while (head != NULL)
4      head = pop_front(head);
5  }
```

link

Summary

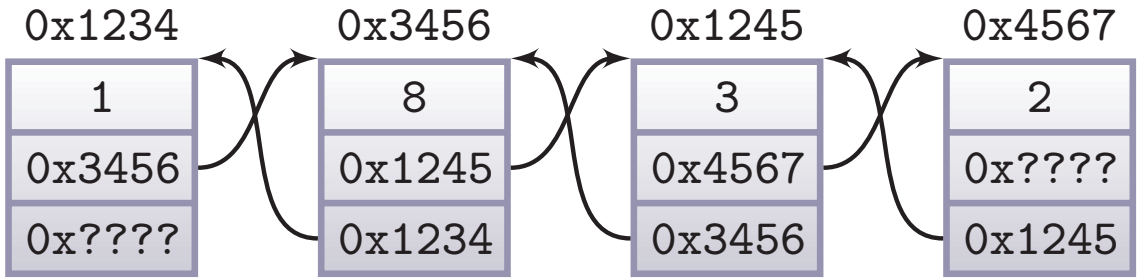
- We have everything we need, but it was really cumbersome, because
  - we can insert element only after (behind) an element
  - we can delete only an element behind another element
  - empty lists and lists with only one element must be handled separately when inserting or deleting

Chapter 2

Doubly linked lists and lists with sentinels

Double linking

- All elements of a doubly linked list contain a pointer to the next and to the previous element too



- Realization in C

```
1  typedef struct listelem {
2    int data;
3    struct listelem *next;
4    struct listelem *prev;
5  } listelem;
```

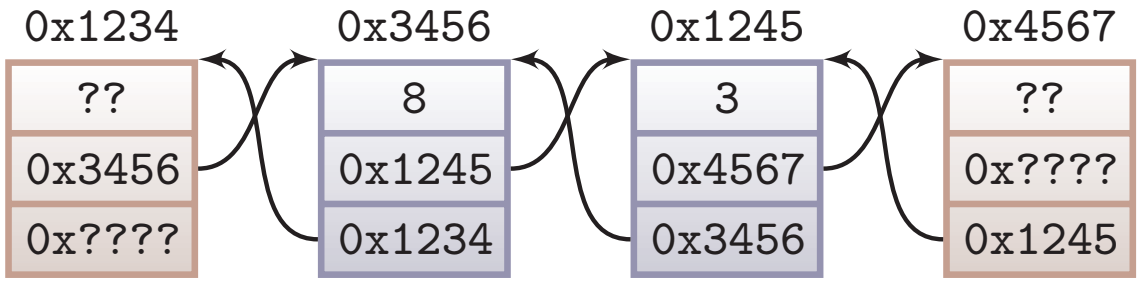
link

- Doubly linking allows us insertion not only behind but also before an element

Sentinels

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel

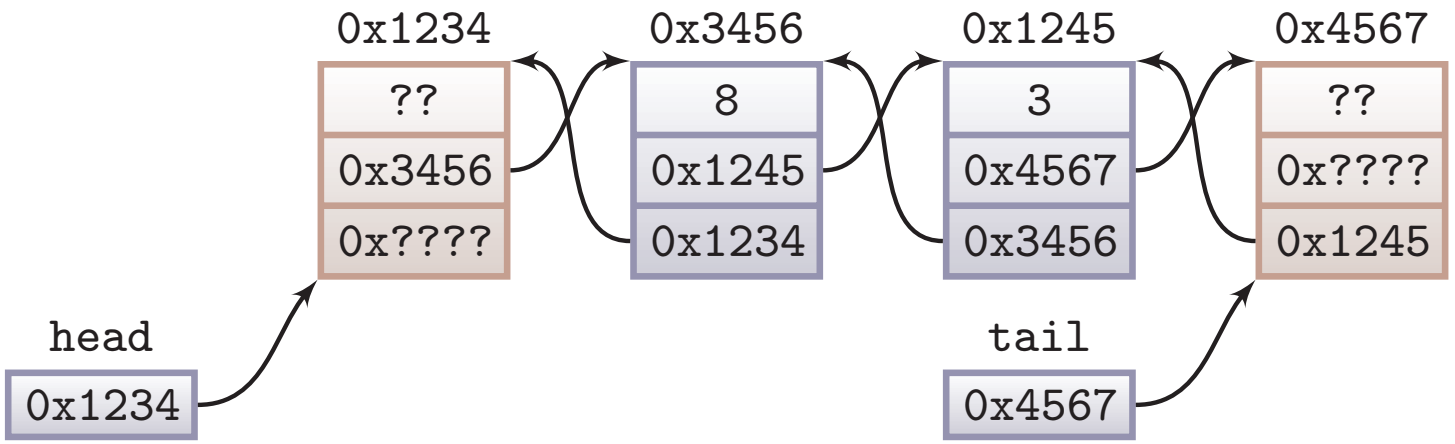


- The type of the sentinel is the same as the type of the intermediate elements
- The data stored in the sentinel is not part of the list
  - many times its value is not concerned (in an unsorted list)
  - in a sorted list the data contained in the sentinel can be the absolutely largest or absolutely smallest element
- Benefits of the list with two sentinels:
  - insertion – even in case of an empty list – is always done between two elements
  - deletion is always done from between two elements

A doubly linked list with two sentinels

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- The sentinels are pointed by the `head` and `tail` pointers



- we enclose these into one entity, this entity will be the list

```
1  typedef struct {
2    listelem *head, *tail;
3  } list;
```

link

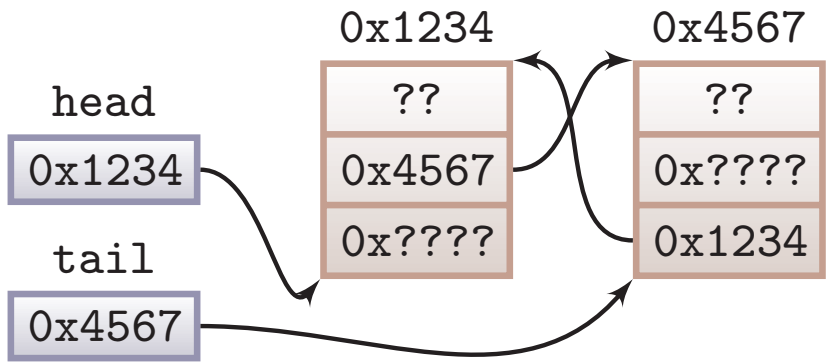
- The sentinels are deleted only when clearing up the list, members of `list` are not changed during the usage of the list

Creating an empty list

- The `create_list` function creates an empty list

```
1  list create_list(void)
2  {
3    list l;
4    l.head = (listelem*)malloc(sizeof(listelem));
5    l.tail = (listelem*)malloc(sizeof(listelem));
6    l.head->next = l.tail;
7    l.tail->prev = l.head;
8    return l;
9  }
```

link



Traversing a list

- The `isempty` function checks whether the list is empty

```
1  int isempty(list l)
2  {
3    return (l.head->next == l.tail);
4  }
```

link

- Traversing a list: with pointer `p` we go from `head->next` to `tail`.

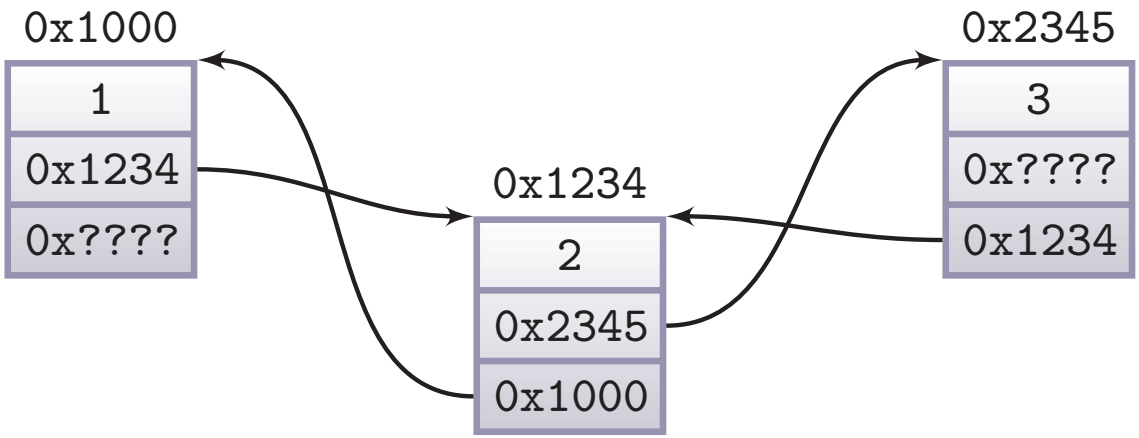
```
1  void print_list(list l)
2  {
3    listelem *p;
4    for (p = l.head->next; p != l.tail; p = p->next)
5      printf("%3d", p->data);
6  }
```

link

Inserting an element between two neighbouring list elements

```
1  void insert_between(listelem *prev, listelem *next,
2    int d)
3  {
4    listelem *p = (listelem*)malloc(sizeof(listelem));
5    p->data = d;
6    p->prev = prev;
7    prev->next = p;
8    p->next = next;
9    next->prev = p;
10 }
```

link



Inserting an element

- to the front of the list

```
1 void push_front(list l, int d) {
2   insert_between(l.head, l.head->next, d);
3 }
```
link

- to the back of the list (we don't check if it is empty)

```
1 void push_back(list l, int d) {
2   insert_between(l.tail->prev, l.tail, d);
3 }
```
link

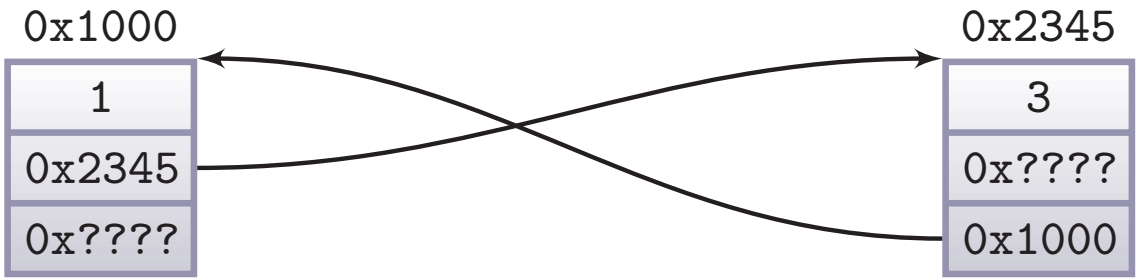
- into a sorted list (we don't need a delayed pointer)

```
1 void insert_sorted(list l, int d) {
2   listelem *p = l.head->next;
3   while (p != l.tail && p->data <= d)
4     p = p->next;
5   insert_between(p->prev, p, d);
6 }
```
link

Deleting an element from a not empty list


```c
void delete(listelem *p)
{
  p->prev->next = p->next;
  p->next->prev = p->prev;
  free(p);
}
```

link



Deleting an element from a list

- from the beginning of the list (the deleted data is returned)

```
1  int pop_front(list l)
2  {
3    int d = l.head->next->data;
4    if (!isempty(l))
5      delete(l.head->next);
6    return d; /* if empty, it returns with
7          sentinel garbage */
8  }
```

link

- from the end of the list

```
1  int pop_back(list l)
2  {
3    int d = l.tail->prev->data;
4    if (!isempty(l))
5      delete(l.tail->prev);
6    return d; /* if empty, it returns with
7          sentinel garbage */
8  }
```

link

Deleting an element from a list

- deleting the selected element

```
1  void remove_elem(list l, int d)
2  {
3    listelem *p = l.head->next;
4    while (p != l.tail && p->data != d)
5      p = p->next;
6    if (p != l.tail)
7      delete(p);
8  }
```

link

- deleting the entire list (also the sentinels)

```
1  void dispose_list(list l) {
2    while (!isempty(l))
3      pop_front(l);
4    free(l.head);
5    free(l.tail);
6  }
```

link

Usage

- A simple application

```
1  list l = create_list();
2  push_front(l, -1);
3  push_back(l, 1);
4  insert_sorted(l, -3);
5  insert_sorted(l, 8);
6  remove_elem(l, 1);
7  print_list(l);
8  dispose_list(l);
```

link

- Of course we can store any data in lists, not only `int` values
- It is useful to separate the stored data and the pointers of the list according to the following

```
1  typedef struct {
2    char name[30];
3    int age;
4    ...
5    double height;
6  } data_t;
7
8  typedef struct listelem {
9    data_t data;
10   struct listelem *next, *prev;
11 } listelem;
```

- If the data stored is a single structure type member, then similarly to the case when having only an `int`, we can use it for assignment of value with only one single instruction, it can be a parameter of a function or a return value.

Chapter 3

Special lists

FIFO

FIFO-buffer

FIFO (First In First Out) – we can access the elements in the order of their insertion

- Typical application: queue, where the elements are processed in the order of their arrival
- Realization: eg. with the previous list.
  - for insertion only `push_front`
  - for taking out only `pop_back`

  functions are used.

Stack

Stack (Stack/LIFO-buffer)

LIFO (Last In First Out) – we can access elements in the reversed order of their insertion

- Typical application: storing "undo"-list, storing return addresses of functions
- Realization: eg. with the previous list.
  - for insertion only `push_front`
  - for taking out only `pop_front`

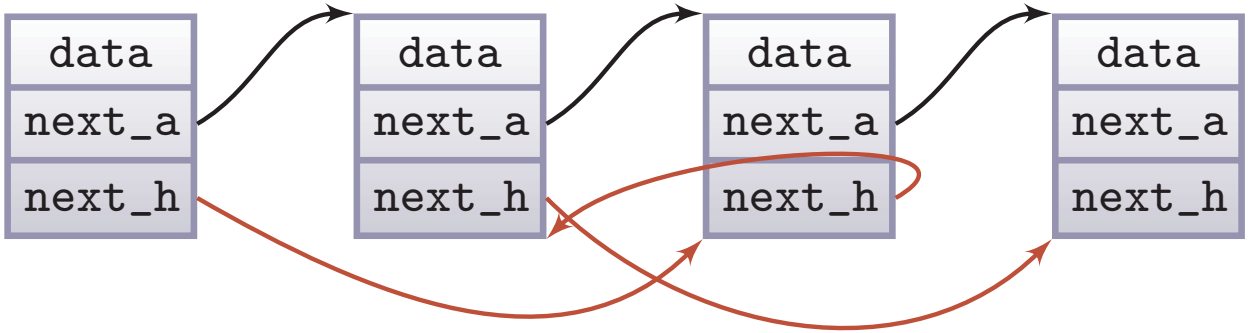
  functions are used.

List sorted in different orders

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- Type for elements of a list sorted in different orders simultaneously

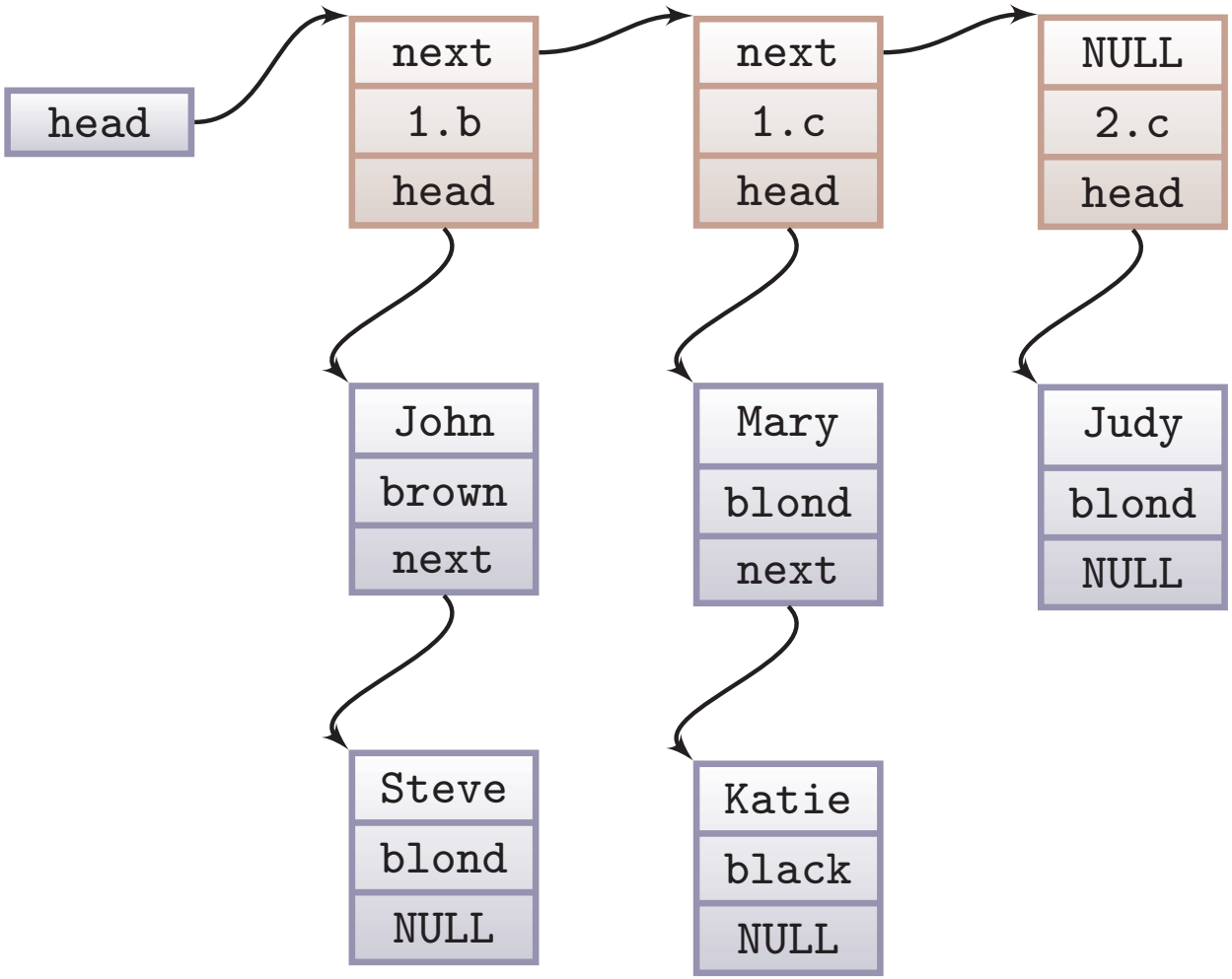
```
1  typedef struct person {
2    data_t data;                    /* data of person */
3    struct person *next_age;        /* next youngest */
4    struct person *next_height;     /* next smallest */
5  } person;
```



Comb list

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

- List of classes, where each class contains the list of the students.



Comb list – declarations

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
typedef struct student_elem {
  char name[50];                /* name */
  colour_t hair_colour;         /* hair colour (typedef) */
  struct student_elem *next; /* linking */
} student_elem;          /* student list element */

typedef struct class_elem {
  char name[10];                /* name of class */
  student_elem *head;           /* list of students */
  struct class_elem *next;      /* linking */
} class_elem;            /* class list element */
```

Comb list – separating data

DEPARTMENT OF NETWORKED SYSTEMS AND SERVICES

```c
typedef struct {
  char name[50];                /* name */
  colour_t hair_colour;         /* hair colour (typedef) */
} student_t;                  /* student data */

typedef struct student_elem {
  student_t student;            /* the student */
  struct student_elem *next; /* linking */
} student_elem;             /* student list element */

typedef struct {
  char name[10];                /* name of class */
  student_elem *head;           /* list of student */
} class_t;                    /* data for class */

typedef struct class_elem {
  class_t class;                /* the class itself */
  struct class_elem *next;      /* linking */
} class_elem;               /* class list element */
```

Thank you for your attention.