

Doubly linked lists – Special lists

Basics of Programming 1



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

9 December, 2020

1 Doubly linked lists and lists with sentinels

- Traversing
- Insertion
- Deletion
- Example

2 Special lists

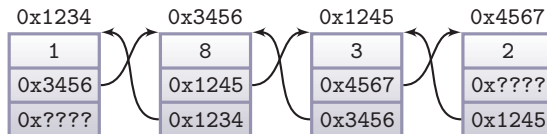
- FIFO
- Stack
- Lists sorted in different orders
- Comb list

Chapter 1

Doubly linked lists and lists with sentinels

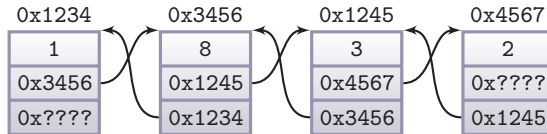
Double linking

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



Double linking

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



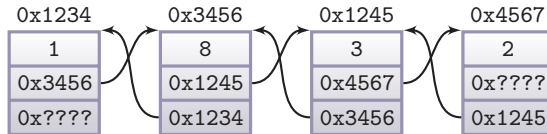
- Realization in C

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

[link](#)

Double linking

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



- Realization in C

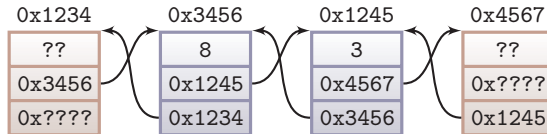
```
1 typedef struct listelem {  
2     int data;  
3     struct listelem *next;  
4     struct listelem *prev;  
5 } listelem;
```

[link](#)

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

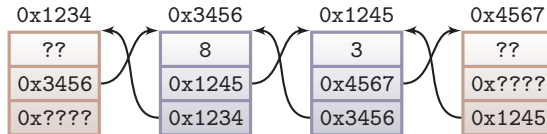
Sentinels

- 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



Sentinels

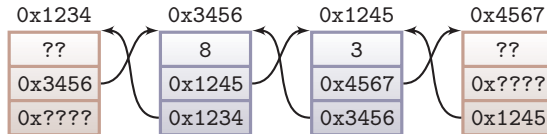
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- The type of the sentinel is the same as the type of the intermediate elements

Sentinels

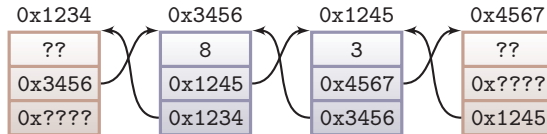
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- 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

Sentinels

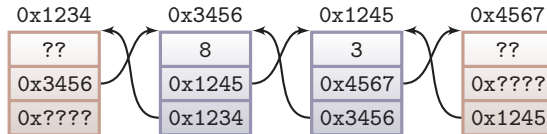
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- 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)

Sentinels

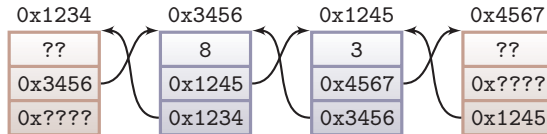
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- 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

Sentinels

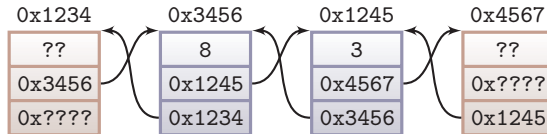
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- 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:

Sentinels

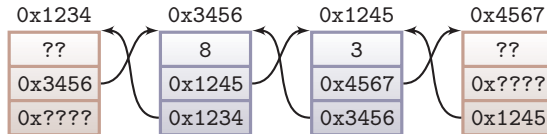
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- 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

Sentinels

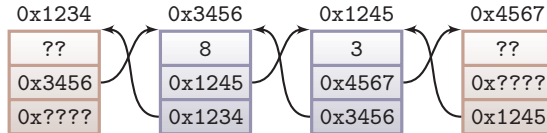
- A list with sentinels means that the list is closed with a non-valid element at one or at both ends, this non-valid element is the sentinel



- 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

Sentinels

- 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

- The sentinels are pointed by the head and tail pointers



A doubly linked list with two sentinels

- 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](#)

A doubly linked list with two sentinels

- 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;

```

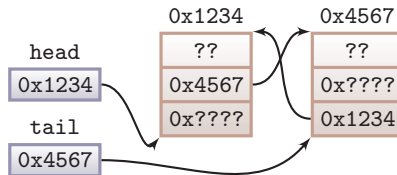
[link](#)

- 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.

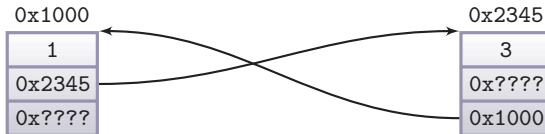
```
1 void print_list(list l)
2 {
3     listelem *p;
4     for (p = l.head->next; p != l.tail; p = p->next)
5         printf("%3d", p->data);
6 }
```

[link](#)

Inserting an element between two neighbouring list elements



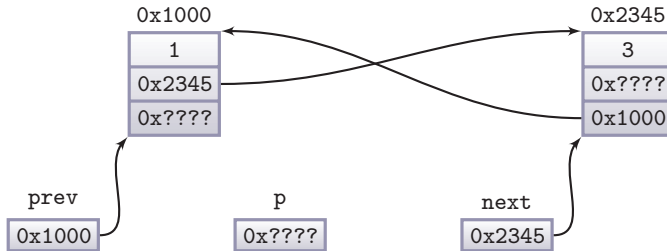
```
1 void insert_between(listelem *prev, listelem *next,  
2     int d)  
3 {  
4     listelem *p = (listelem*)malloc(sizeof(listelem));  
5     p->data = d;  
6     p->prev = prev;  
7     prev->next = p;  
8     p->next = next;  
9     next->prev = p;  
10 }
```



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 }
  
```

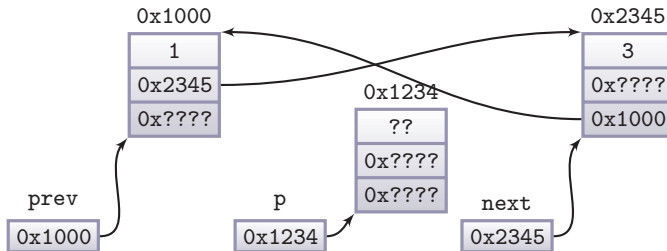


Inserting an element between two neighbouring list elements

```

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 }

```



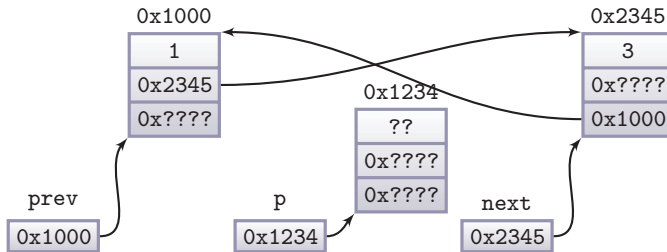
Inserting an element between two neighbouring list elements



```

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 }

```



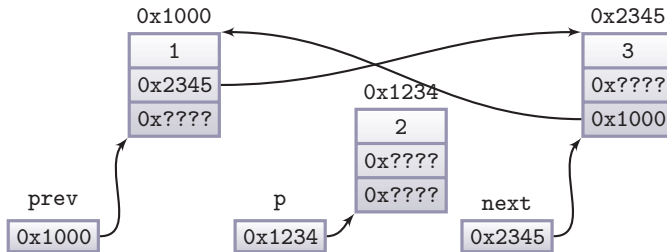
Inserting an element between two neighbouring list elements



```

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 }

```

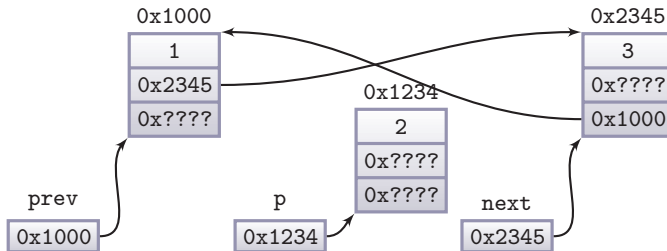


Inserting an element between two neighbouring list elements

```

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 }

```

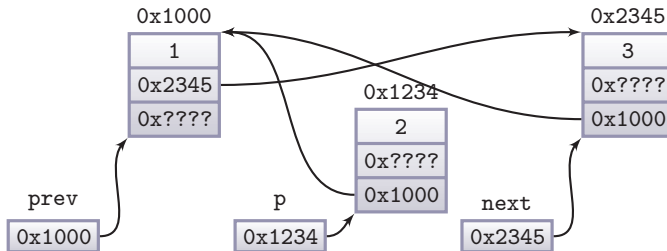


Inserting an element between two neighbouring list elements

```

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 }

```



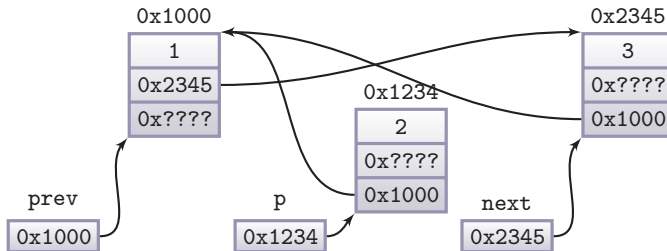
Inserting an element between two neighbouring list elements



```

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 }

```

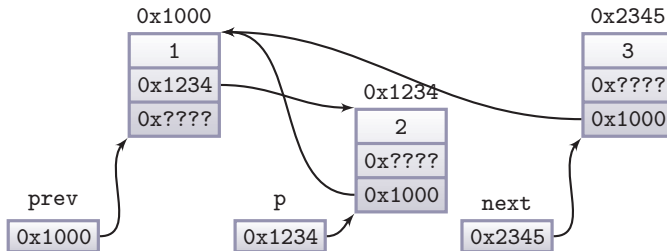


Inserting an element between two neighbouring list elements

```

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 }

```



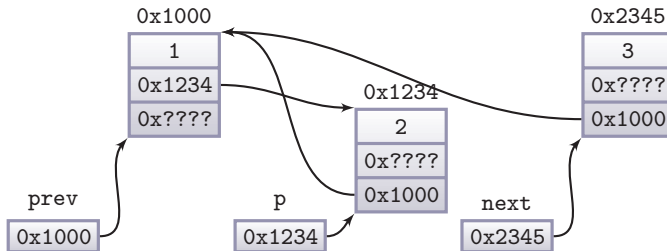
Inserting an element between two neighbouring list elements



```

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 }

```



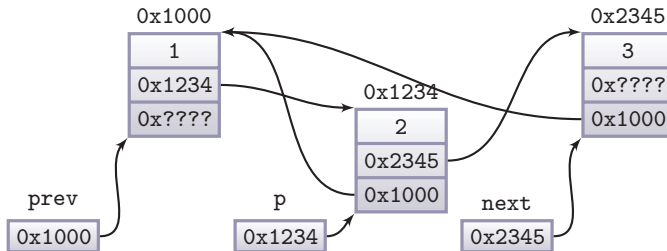
Inserting an element between two neighbouring list elements



```

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 }

```

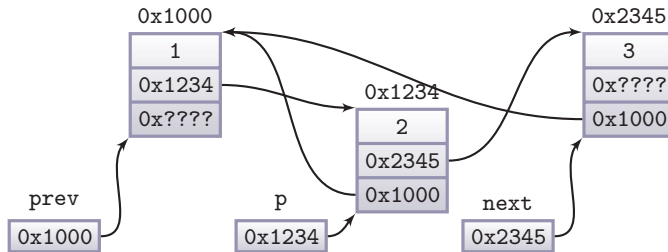


Inserting an element between two neighbouring list elements

```

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 }

```

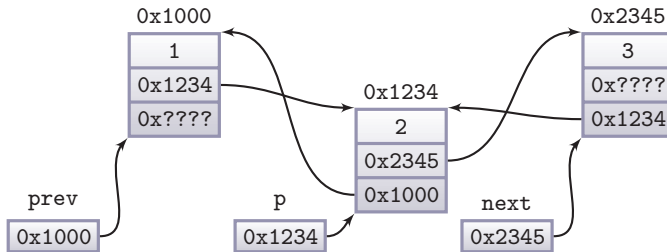


Inserting an element between two neighbouring list elements

```

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 }

```



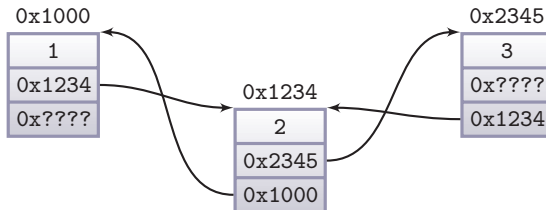
Inserting an element between two neighbouring list elements



```

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](#)

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](#)

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](#)

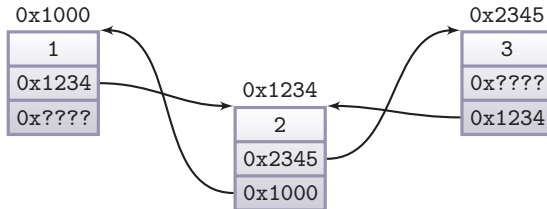
- into a sorted list (we don't need a delayed pointer)

```
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](#)

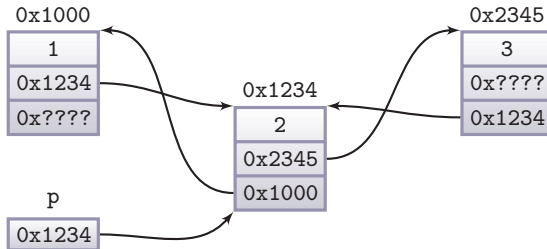
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



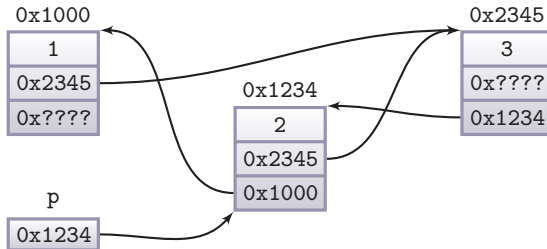
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



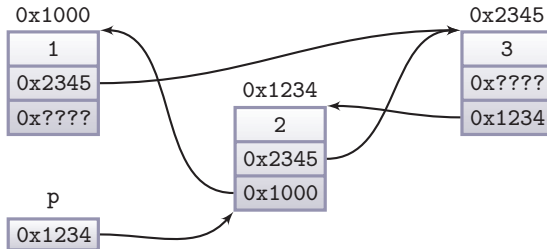
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



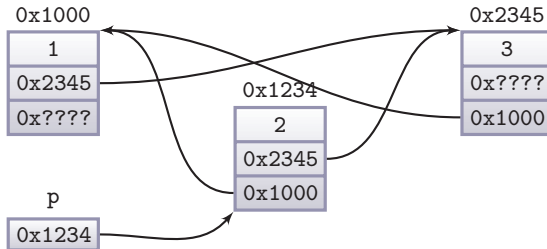
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



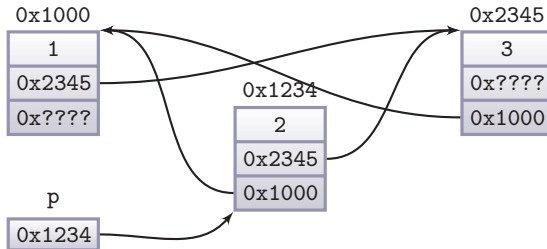
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



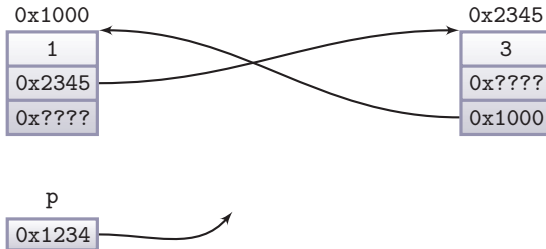
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



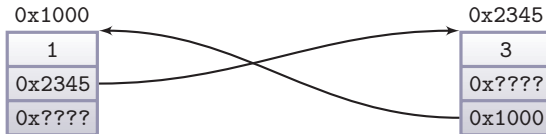
Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```



Deleting an element from a not empty list

```
1 void delete(listelem *p)
2 {
3     p->prev->next = p->next;
4     p->next->prev = p->prev;
5     free(p);
6 }
```

[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](#)

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  int main(void)
2  {
3      list l = create_list();
4      push_front(l, -1);
5      push_back(l, 1);
6      insert_sorted(l, -3);
7      insert_sorted(l, 8);
8      remove_elem(l, 1);
9      print_list(l);
10     dispose_list(l);
11     return 0;
12 }
```

[link](#)

- Of course we can store any data in lists, not only `int` values

- 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;
```

- 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 2

Special lists

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/LIFO-buffer)

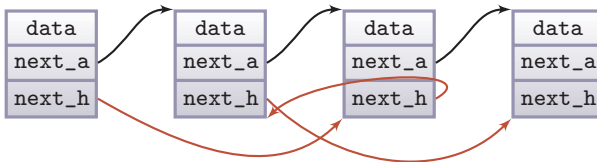
LIFO (Last In First Out) – we can access elements in the reversed order of their insertion

- 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

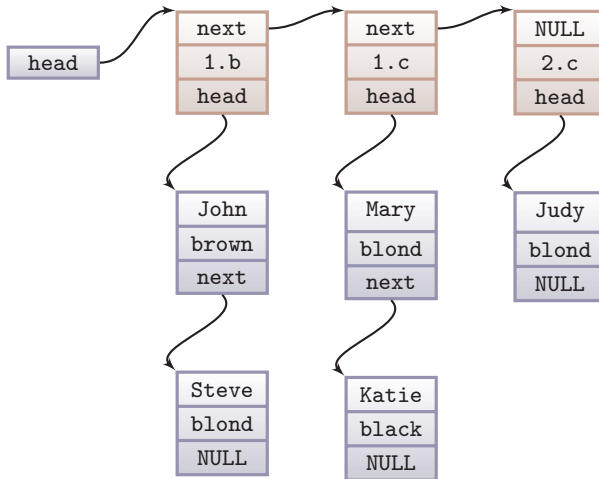
- Type for elements of a list sorted in different orders simultaneously

```
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

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



Comb list – declarations

```
1 typedef struct student_elem {
2     char name[50];           /* name */
3     colour_t hair_colour;    /* hair colour (typedef) */
4     struct student_elem *next; /* linking */
5 } student_elem;             /* student list element */
6
7 typedef struct class_elem {
8     char name[10];           /* name of class */
9     student_elem *head;      /* list of students */
10    struct class_elem *next;   /* linking */
11 } class_elem;               /* class list element */
```

Comb list – separating data

```
1 typedef struct {
2     char name[50];           /* name */
3     colour_t hair_colour;    /* hair colour (typedef) */
4 } student_t;                /* student data */
5
6 typedef struct student_elem {
7     student_t student;      /* the student */
8     struct student_elem *next; /* linking */
9 } student_elem;            /* student list element */
10
11 typedef struct {
12     char name[10];          /* name of class */
13     student_elem *head;     /* list of student */
14 } class_t;                 /* data for class */
15
16 typedef struct class_elem {
17     class_t class;          /* the class itself */
18     struct class_elem *next; /* linking */
19 } class_elem;              /* class list element */
```

Thank you for your attention.