

The background features a white surface with scattered, colorful abstract shapes. These include thin, curved lines in shades of light blue, light green, and light purple. Interspersed among these are numerous small, yellow, triangular shapes that resemble confetti or paper scraps. The overall aesthetic is clean, modern, and celebratory.

# **C Programming Basic – week 3**


# Topics

- Self referential structure in C
- Data structure “single linked LIST”
  - Implementation of single linked LIST
  - Algorithm for scanning data
  - Algorithm for inserting, deleting
- Data structure “double linked LIST”
  - Implementation of double linked LIST
  - Algorithm for scanning data
  - Algorithm for inserting, deleting

# Self-Referential Structures

- One or more of its components is a pointer to itself.

```
struct list {  
    char data;  
    struct list *link;  
};
```

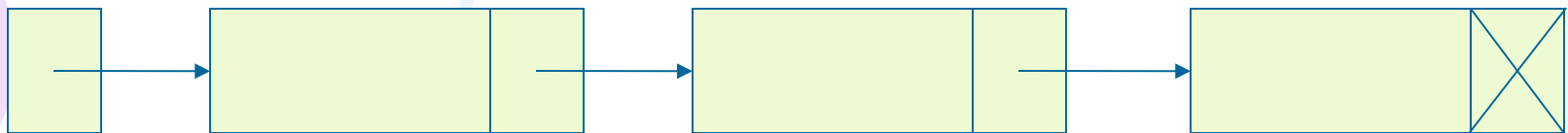


```
list item1, item2, item3;  
item1.data='a';  
item2.data='b';  
item3.data='c';  
item1.link=item2;  
item2.link=item3;  
item3.link=NULL;
```



# Implementation of List in C

- "LIST" means data structure that keeps the information of the location of next element generally.
- The elements of "Single linked LIST" have only next location.
- In C, the pointer is used for the location of the next element.
- Array: We can access any data immediately.
- Linked List: We can change the number of data in it.



root (or head)

NULL<sub>4</sub>

# Declaration of a Linked List

```
typedef ...  
    elementtype;  
typedef struct node{  
    elementtype element;  
    node* next;  
};  
node* root;  
node* cur;
```

```
typedef ...  
    elementtype;  
struct node{  
    elementtype element;  
    struct node* next;  
};  
struct node* root;  
struct node* cur;
```

# Memory allocation for an element

- We need to allocate a memory bloc for each node (element) via a pointer.

```
struct node * new;
```

```
new = (struct node*) malloc(sizeof(struct node));
```

```
new->element = ...
```

```
new->next = null;
```

- new->addr means (\*new).addr.
- “pointer variable for record structure” -> “member name”

# Example

- We are now designing “address list” for mobile phones.
- You must declare a record structure that can keep a name, a phone number, and a e-mail address at least.
- And you must make the program which can deals with any number of the data

# Example (cont)

- Create a singly linked list to store a list of phone address.
- Write a function to insert to a list a new element just after the current element and use it to add node to the list
- Write a function for traversing the list to print out all information stored.
- Write a function for the removal of a node in the list.



# Hint

- you can organize elements and data structure using following record structure AddressList. Define by your self a structure for storing information about an address.

```
struct AddressList {  
    struct AddressList *next;  
    struct Address addr;  
};
```

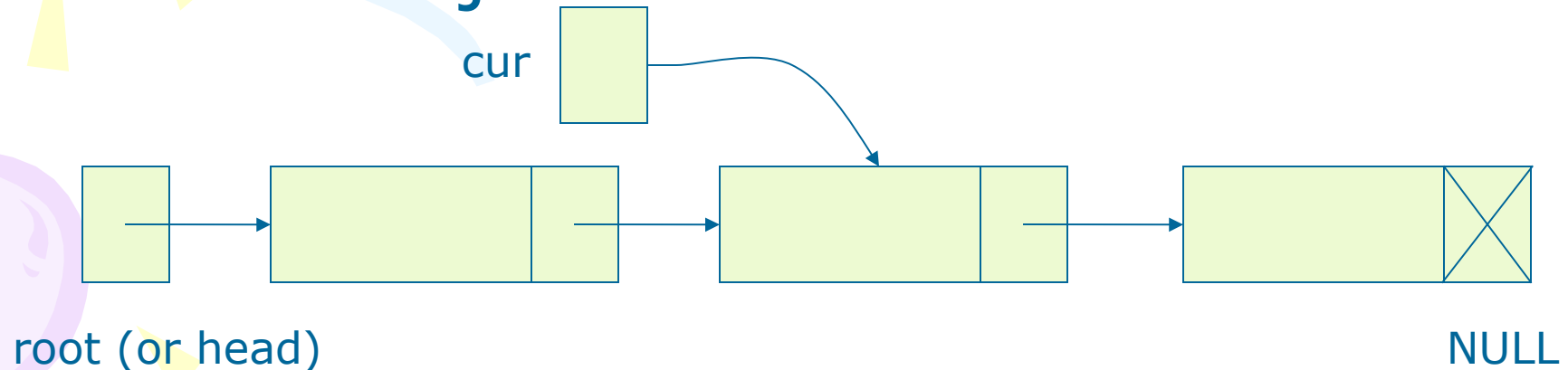
# Declaration of record structure

```
struct AddressList {  
    struct AddressList *next;  
    struct Address addr;  
};
```

- “next” is the pointer variable which can express the next element; an element of AddressList.
- “addr” is instance of an address.

# Important 3 factors of a LIST

- Root: It keeps the head of the list.
- NULL: The value of pointer. It means the tail of the list.
- Cur: Pointer variable that keeps the element just now.



# Initialisation

```
struct AddressList {  
    struct AddressList *next;  
    struct Address addr;  
};  
struct AddressList * root, *cur;
```

# Make new node

```
struct AddressList* makeNewNode() {
    struct AddressList * new = (struct AddressList
        * ) malloc(sizeof(struct AddressList));
    strcpy((new->addr).name, « Tran Van Thanh »);
    ...
    new->next =NULL;
    return new;
}

root = makeNewNode();
cur = root;
```

# Attention

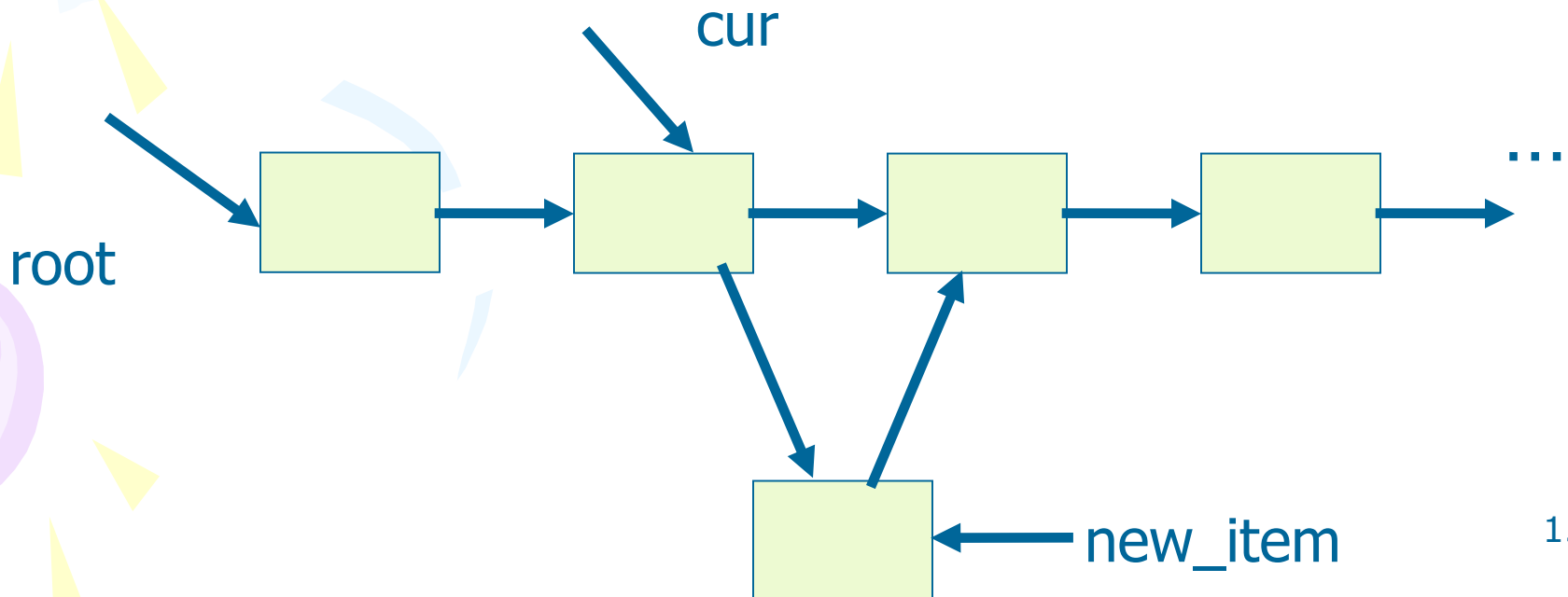
- You can modify the makeNewNode function to receive the data field as parameter:

```
struct AddressList* makeNewNode (Address
    addr) {
    struct AddressList * new = (struct
        AddressList * ) malloc(sizeof(struct
            AddressList));
    new->addr=addr;
    new->next =NULL;
    return new;
}
```

# Link list: insertion

- Just after the current position

```
create new_item  
new->next = cur->next;  
cur->next = new;  
cur = cur->next;
```



# Link list: insertion

- Just after the current position

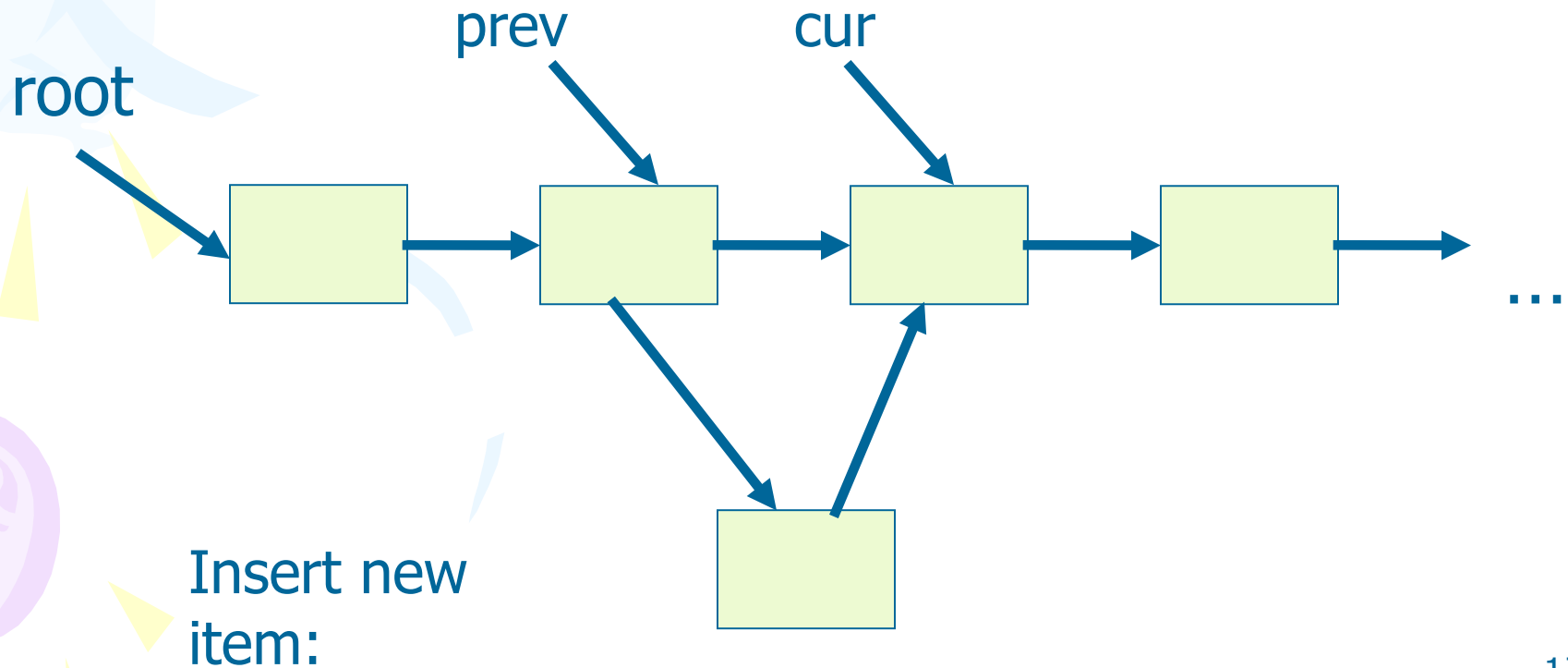
```
new = ( struct AddressList * ) malloc( sizeof( struct AddressList
) );
new->addr = addr;
new->next = NULL;
// new = makeNewNode();
if ( root == NULL ) {
    /* if there is no element */
    root = new;
    cur = root;
} else {
    new->next=cur->next;
    cur->next = new;
    // prev=cur;
    cur = cur->next;
}
}
```

...



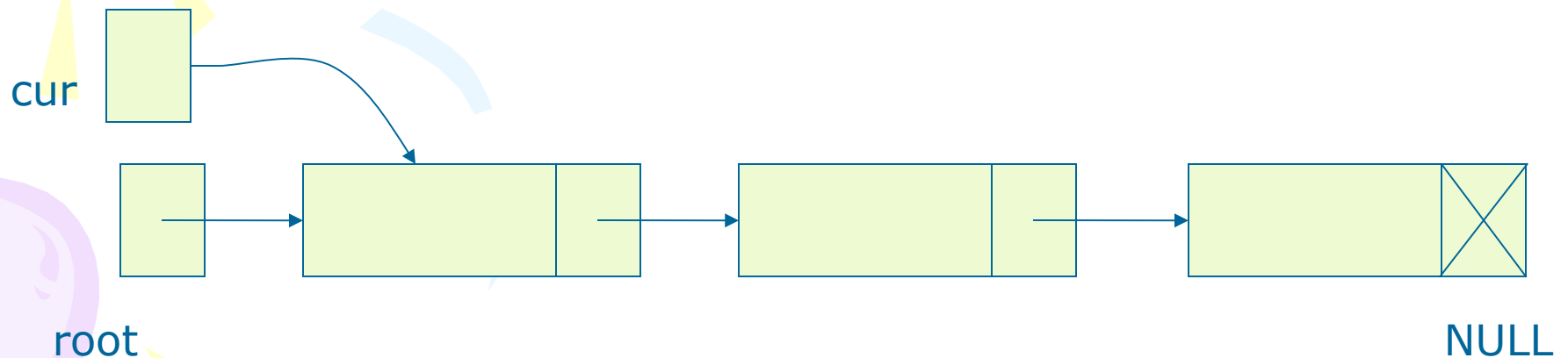
# Linked lists – insertion

- Another case: before the current position



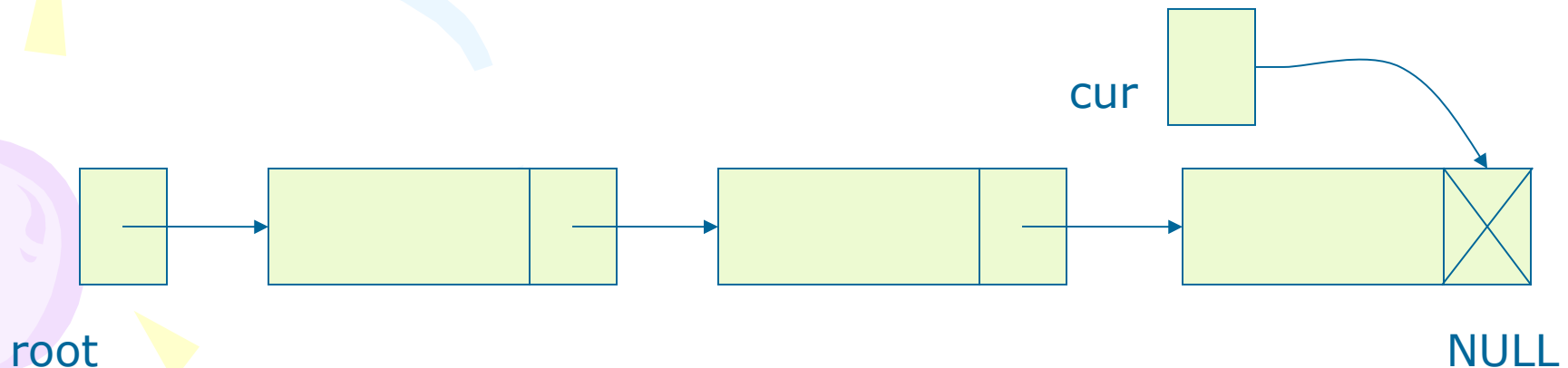
# Traversing a list

```
for ( cur = root; cur != NULL; cur = cur->next ) {  
    showAddress( cur->addr, stdout );  
}
```



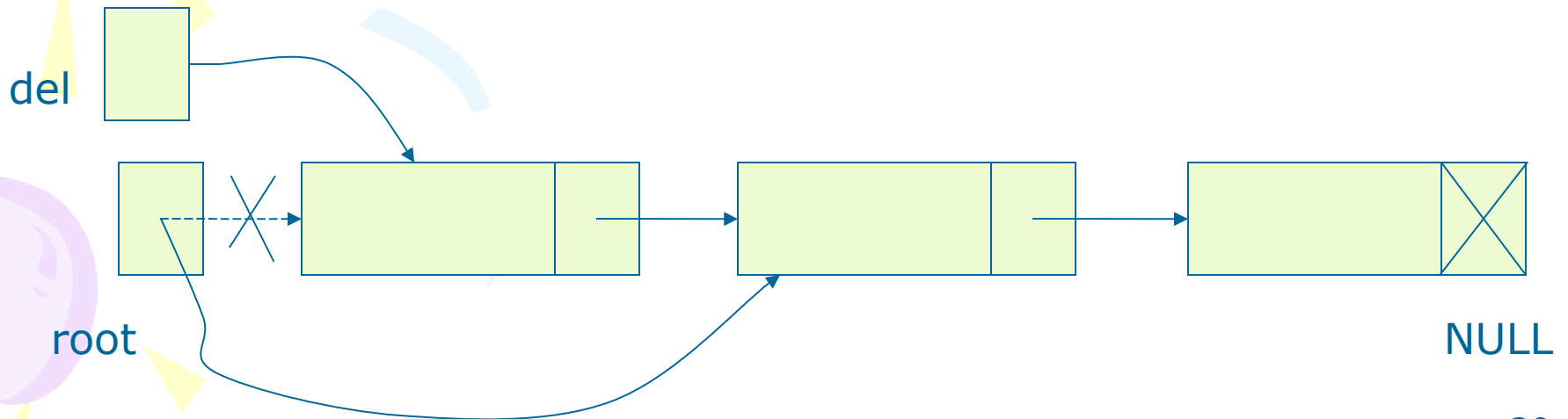
# Traversing a list

- Changing the value of pointer variable cur in sequence.
- These variables are called "iterator."
- The traversing is finished if the value is NULL



# Deletion

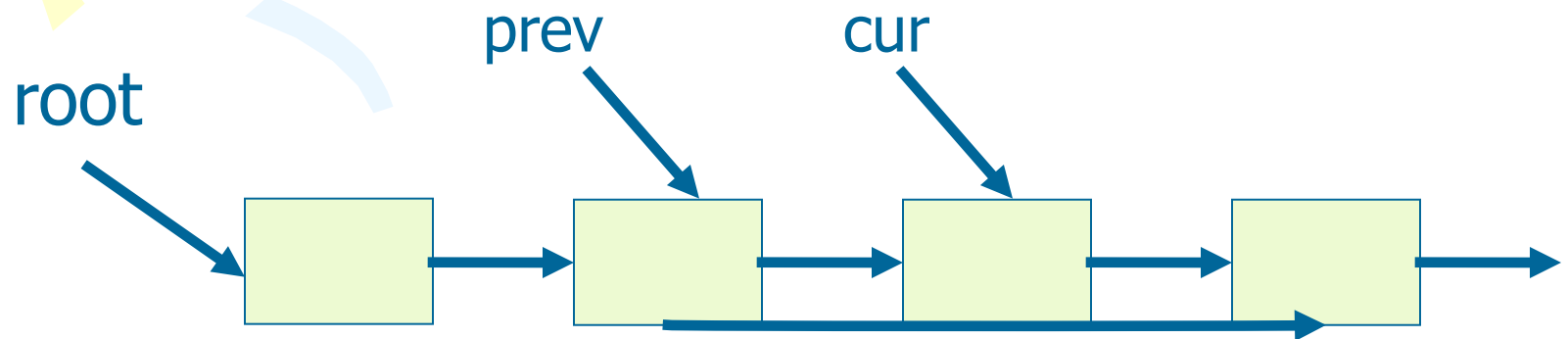
- When we remove the first element  
`root = del->next; free(del);`
- When we remove the first element, change the value of "root" into the value of "next" which is pointed by "del."



# Deletion from the middle

- We want to remove the node pointed by cur
- Determine prev which point to the node just before the node to delete

```
prev->next = cur->next;  
free(cur);
```



# Exercise 3.1

- Read data from file 'phone.dat' (from previous exercises) and load to a single linked list.
- Use functions studied in class
- Print out list of phone number
- Ask user to input new phone number and update to the list

# Exercise 3.2

- Implement function insert, delete with a parameter n (integer) indicating the position of node to be affected.
  - The head position means 0th.
  - 1st means that we want to add the element into the next place of the first element.
  - 2nd means the next place of the second element.

```
struct AddressList *insert (struct AddressList  
    *root, struct Address ad, int n);
```

```
struct AddressList *delete(struct AddressList  
    *root, int n);
```

# Exercise 3.3

- Add function to Exercise 1 to update and delete element given position
- Use functions for deletion in class
- Implement program in menu that search by
  - Phone number
  - Name
- Catch all possible errors
- Add function to reverse the list



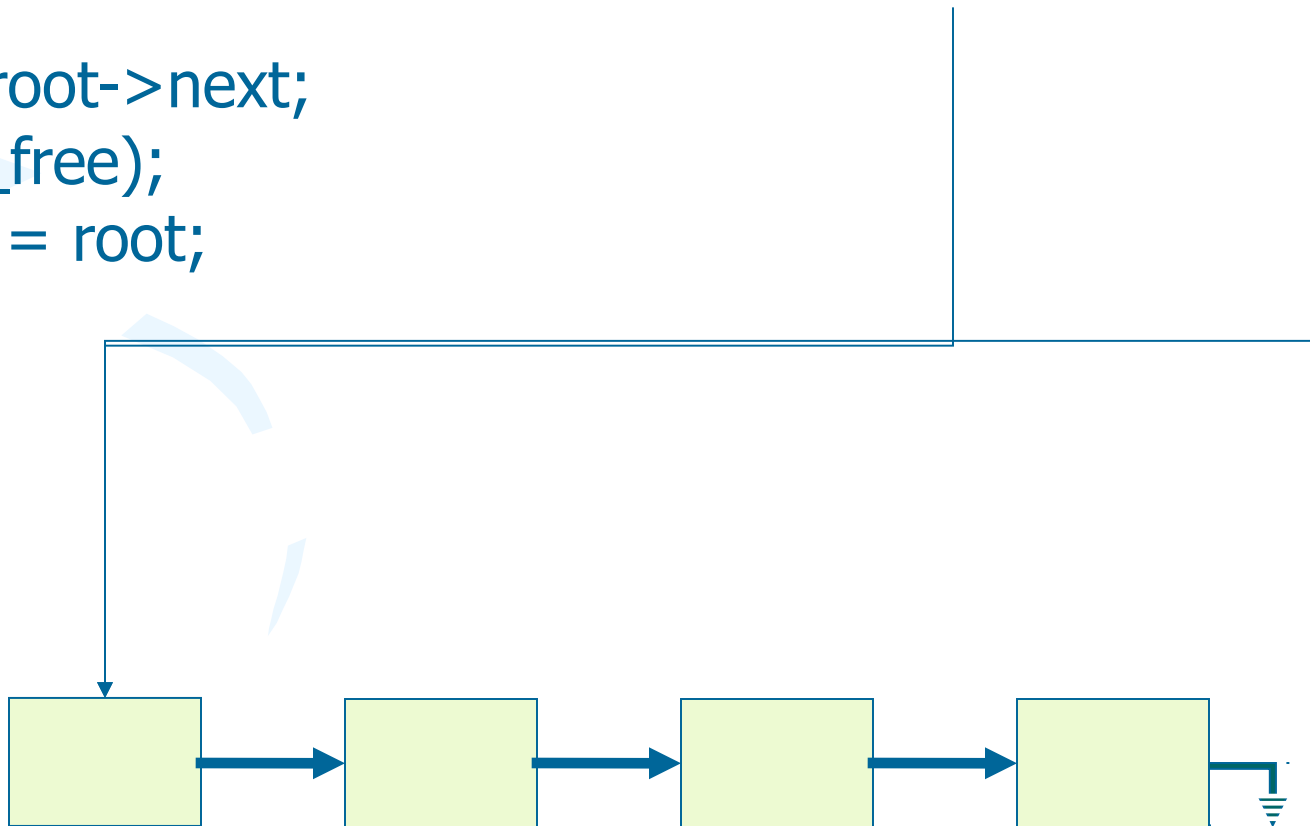
# Exercise 3.4

- a) Write a function to split the list
  - Split into two sub-lists. Input parameter is  $n$ . (starting from 0)
  - Extract – parameters  $n1$  and  $n2$ : Extract a sub-list of  $n2$  elements from position  $n1$ .
  - Add to program menu.
- b) Write a function to write list to a text file. Parameters are root node of the list and file path. Use it to store result of split/extract function.
- c) List data – NokiaDB exercise

# Freeing a list

```
to_free = root ;  
while (to_free != NULL)  
{  
    root = root->next;  
    free(to_free);  
    to_free = root;  
}
```

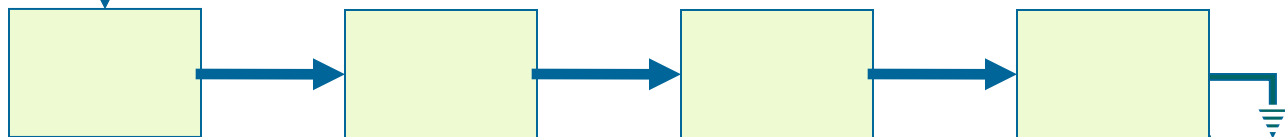
to\_free      root



# Freeing all nodes of a list

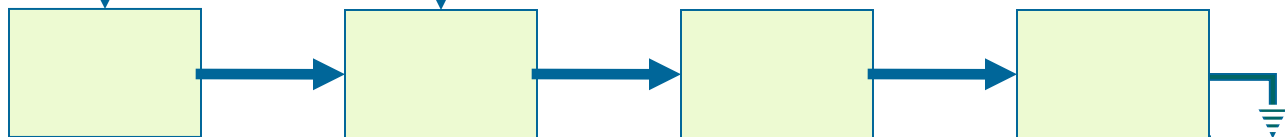
```
to_free = root ;  
while (to_free != NULL)  
{  
  root = root->next;  
  free(to_free);  
  to_free = root;  
}
```

to\_free      root



# Freeing all nodes of a list

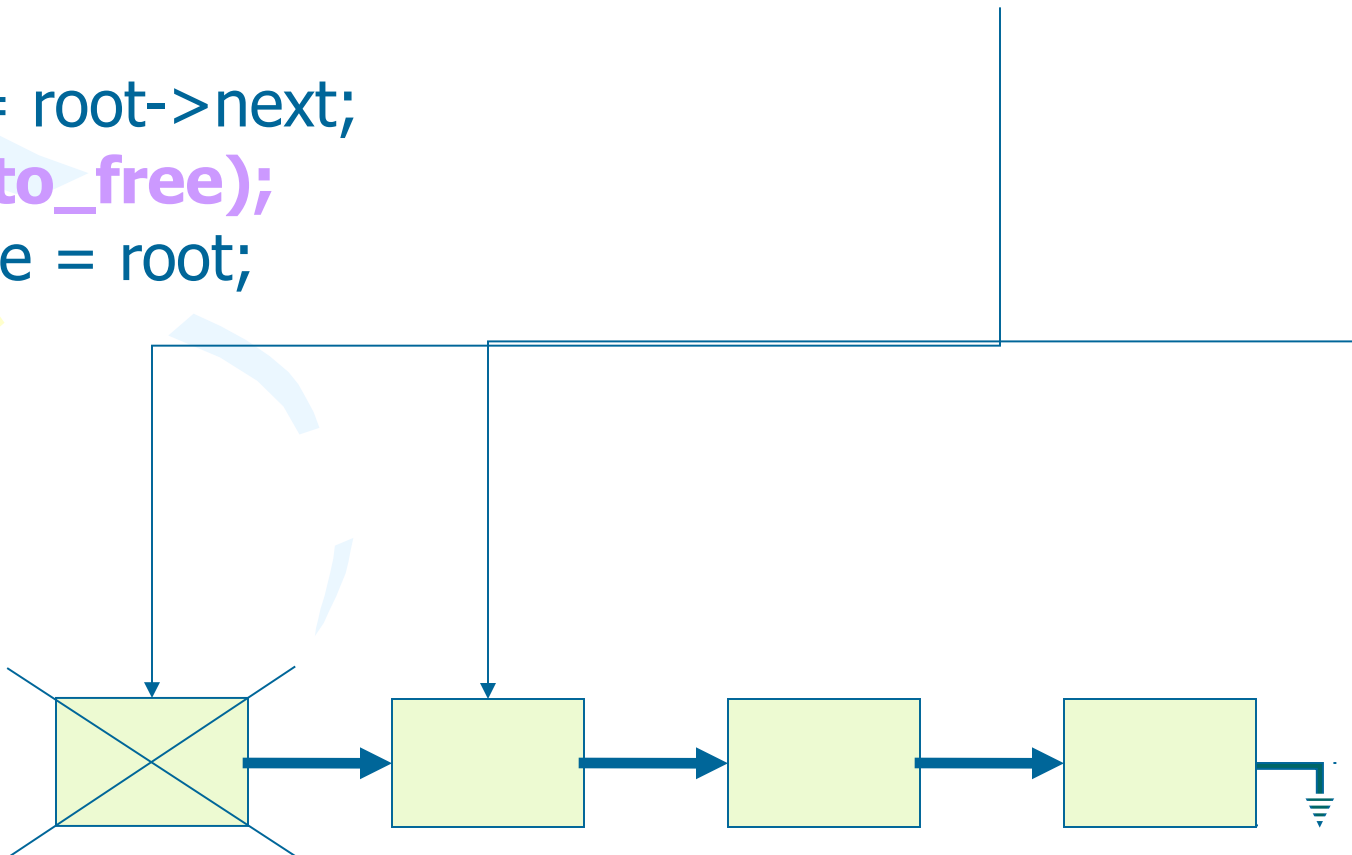
```
to_free = root ;  
while (to_free != NULL)  
{  
    root = root->next;  
    free(to_free);  
    to_free = root;  
}
```



# Freeing all nodes of a list

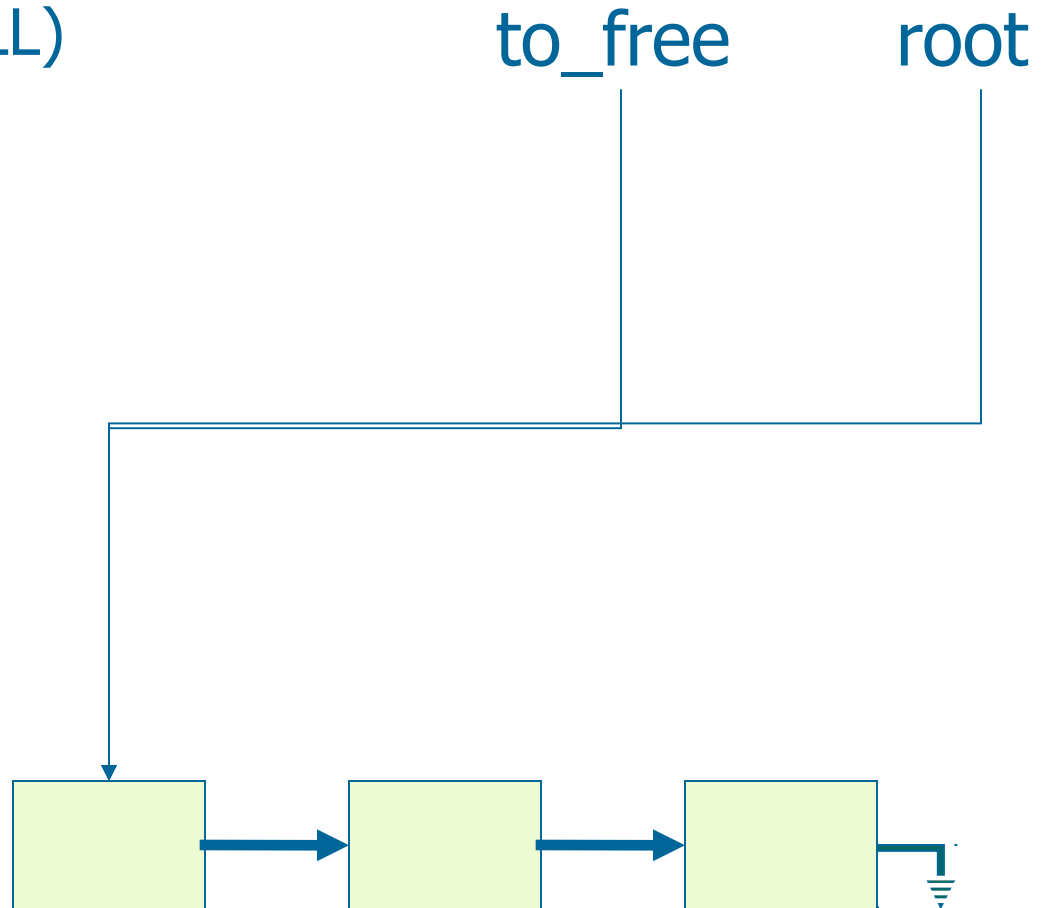
```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```

to\_free      root



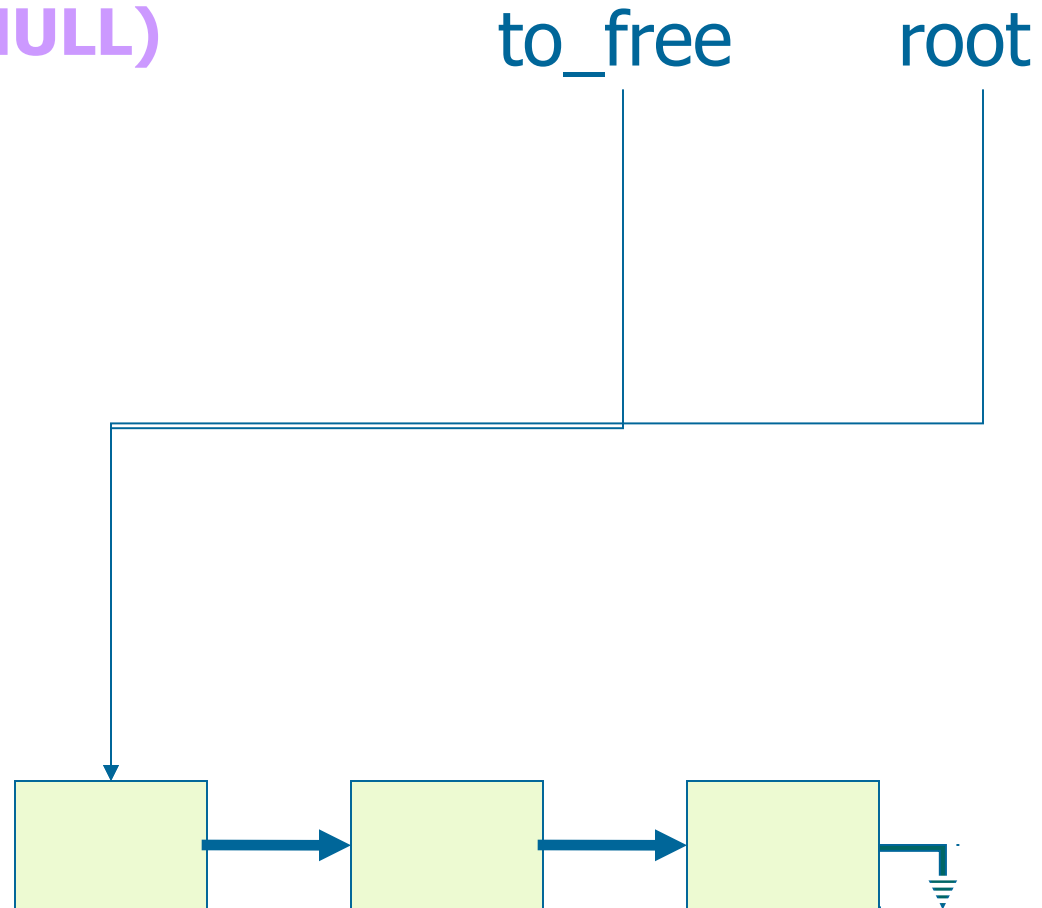
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



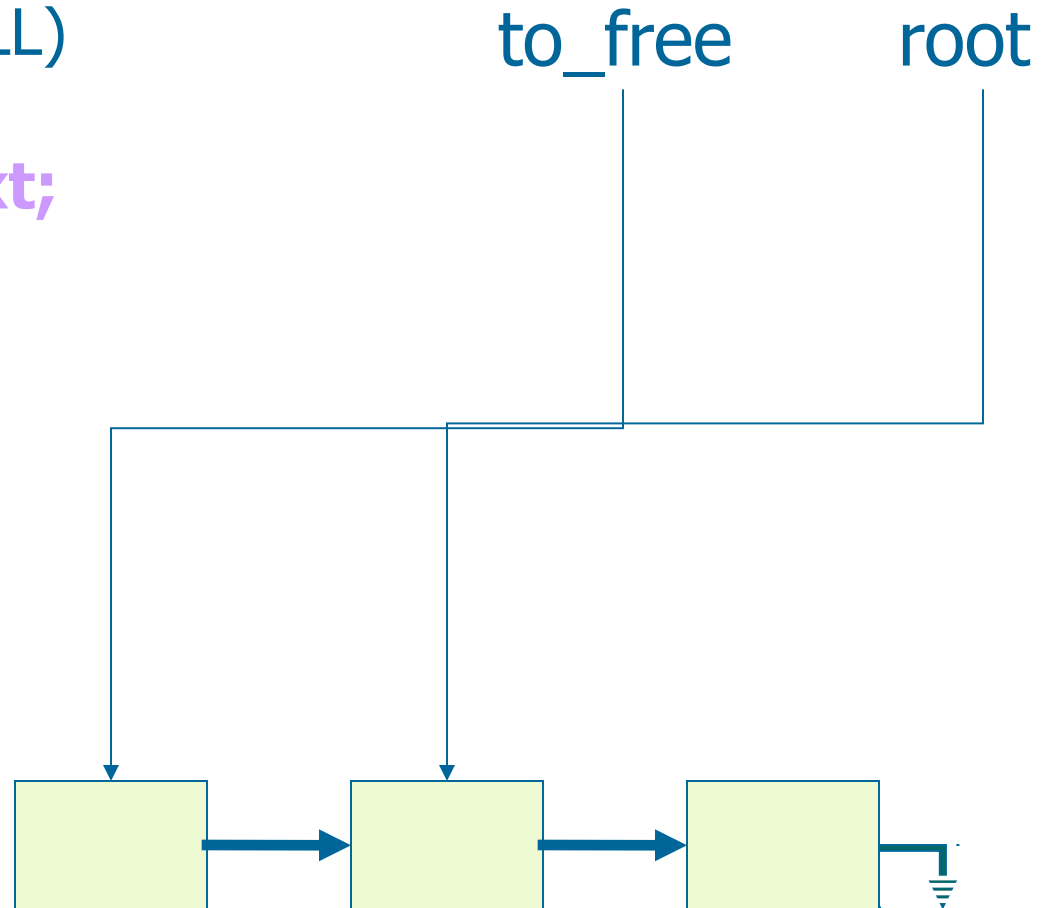
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



# Freeing all nodes of a list

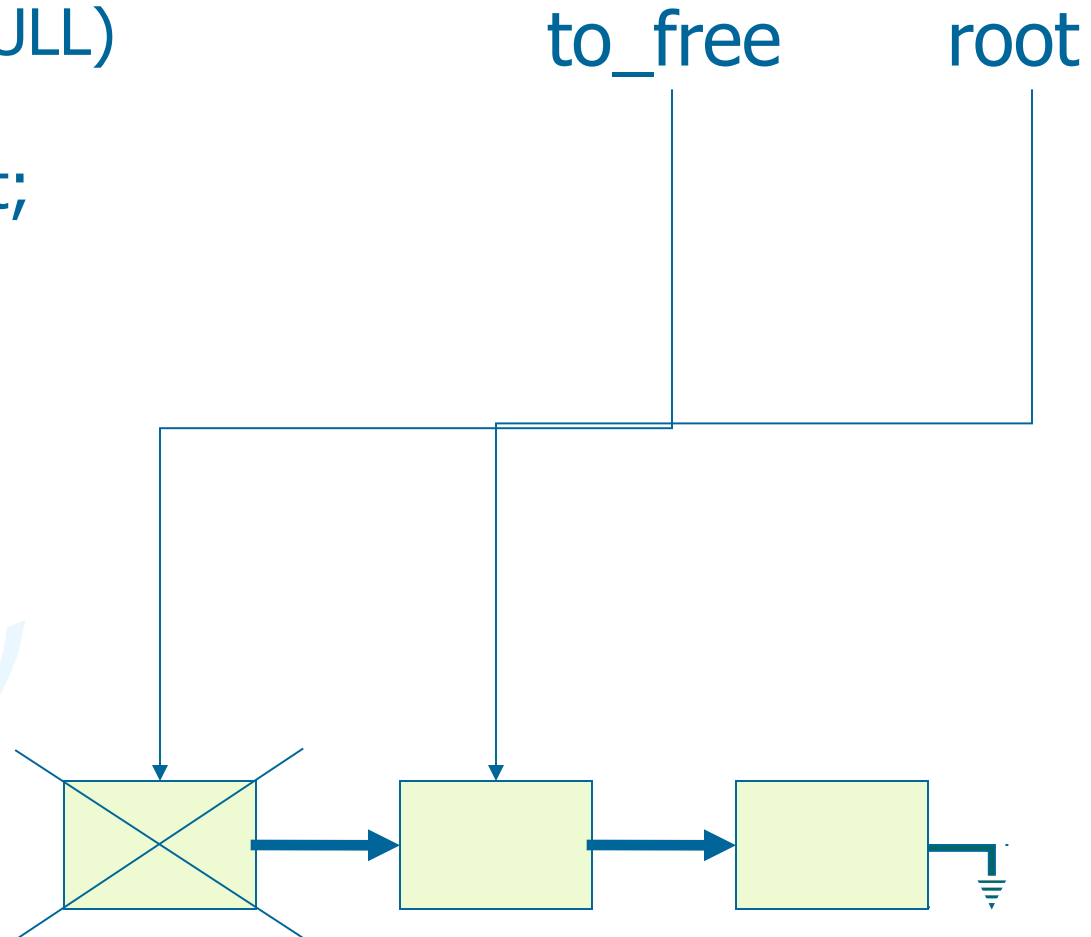
```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```





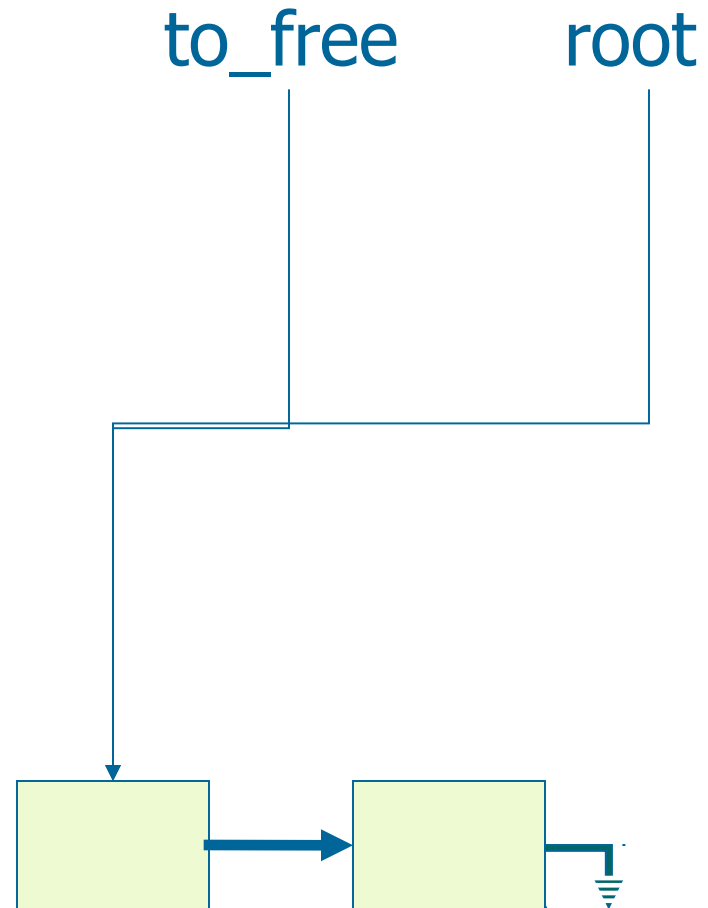
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



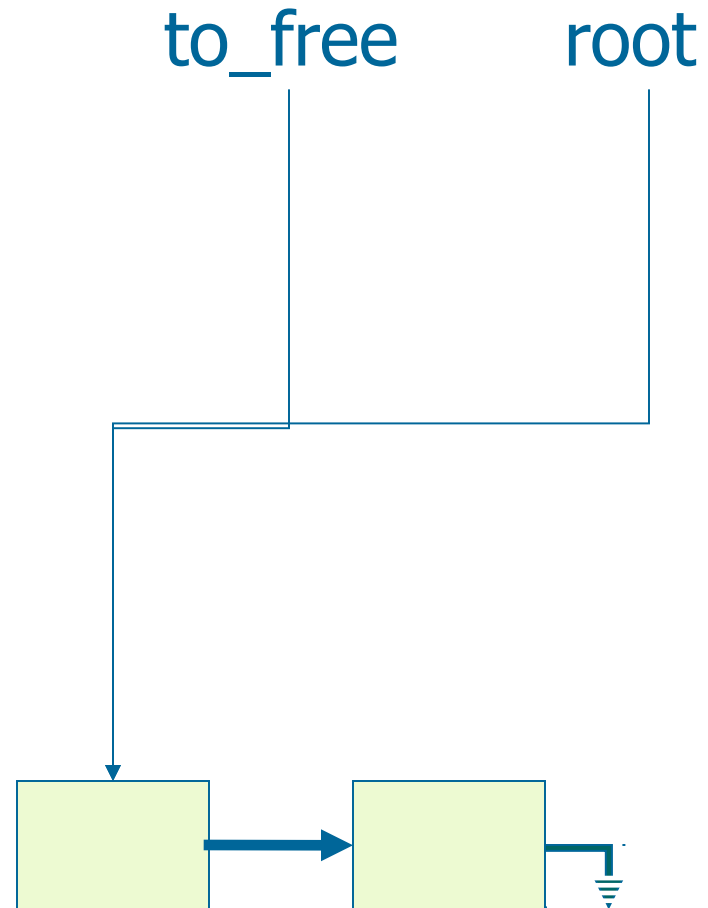
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



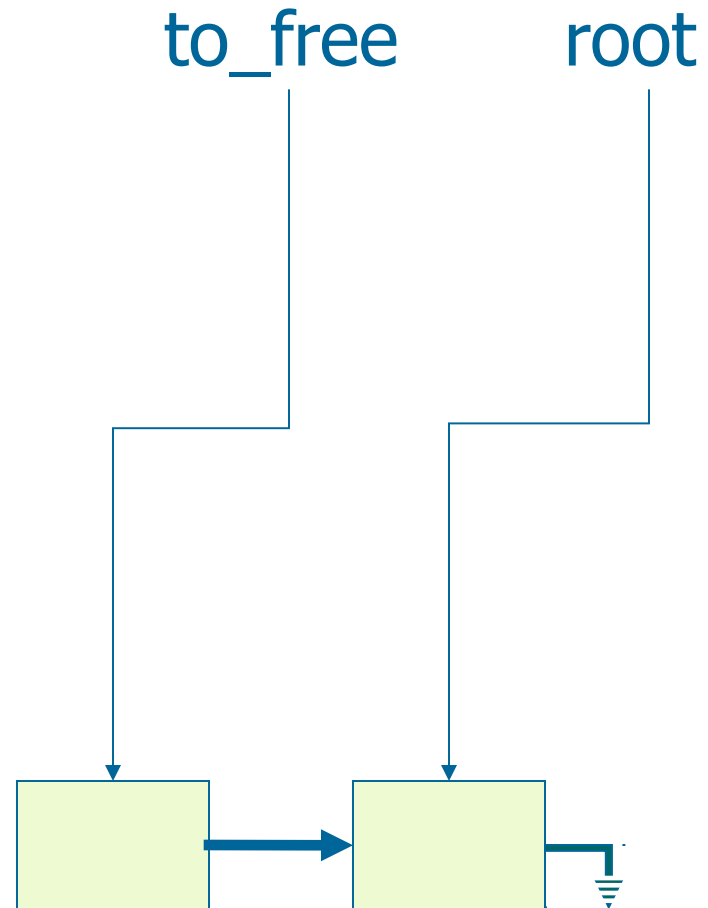
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



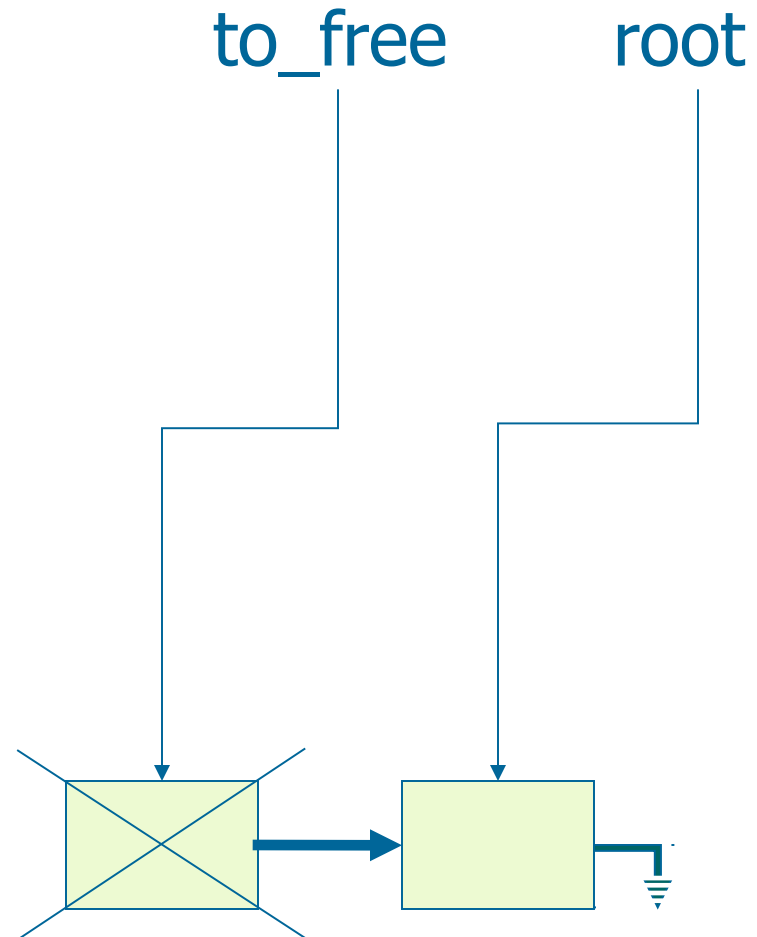
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



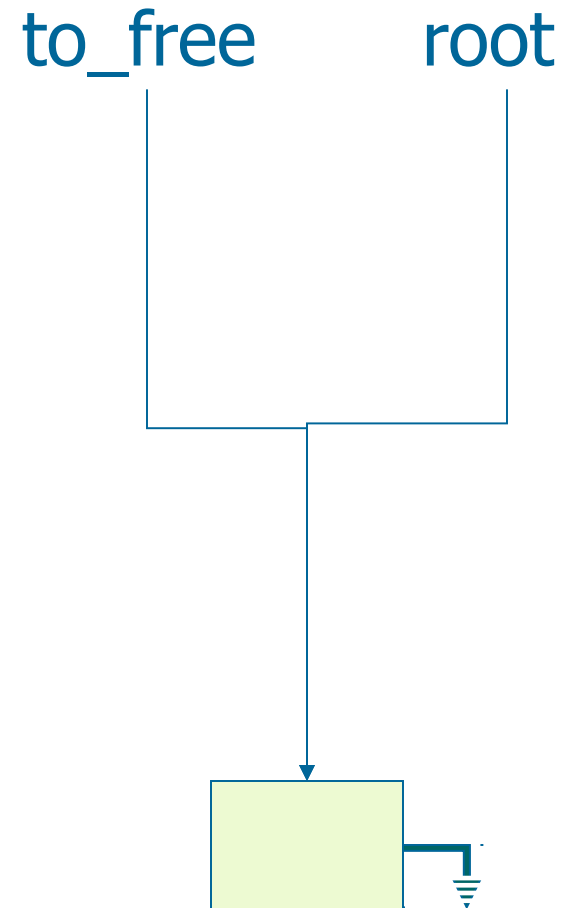
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



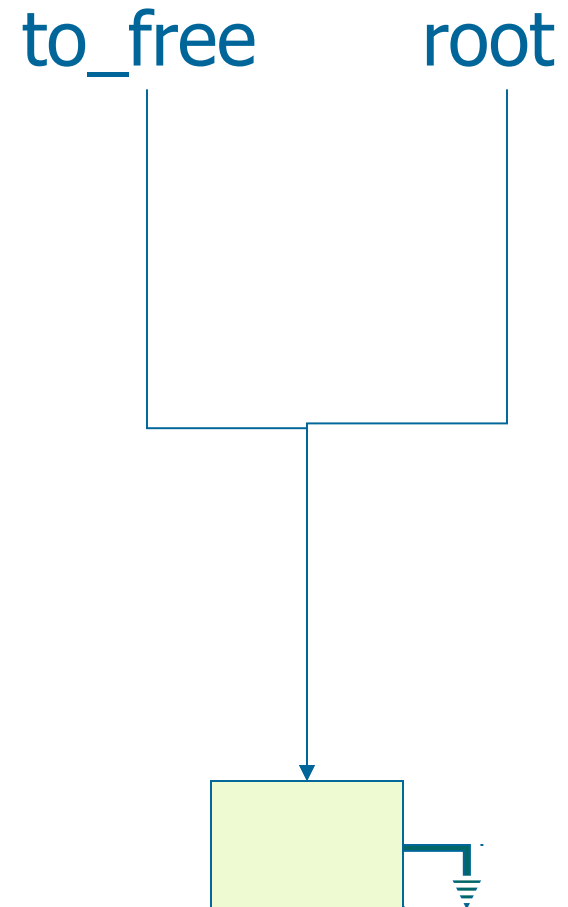
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



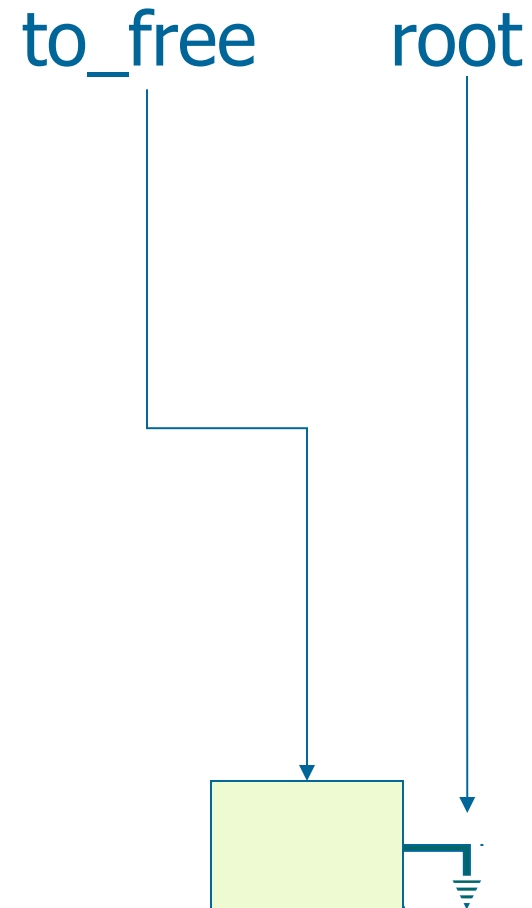
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



# Freeing all nodes of a list

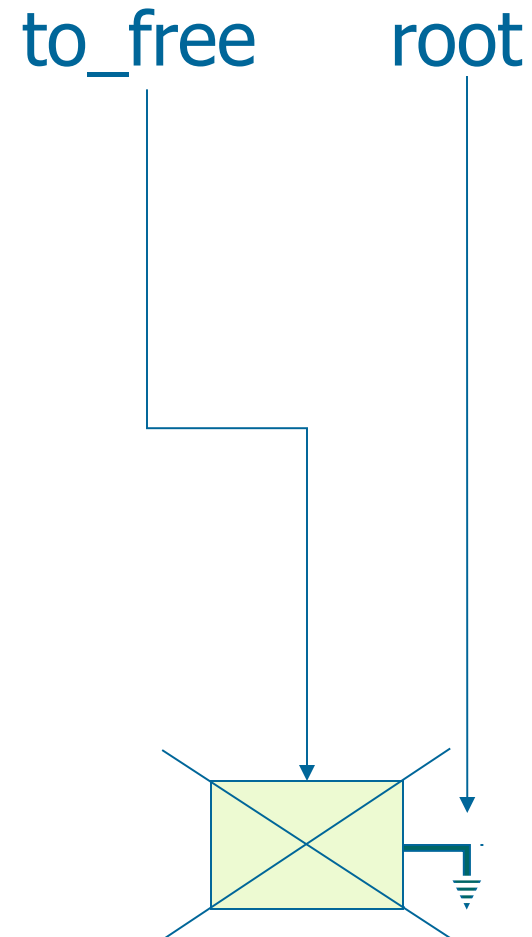
```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```





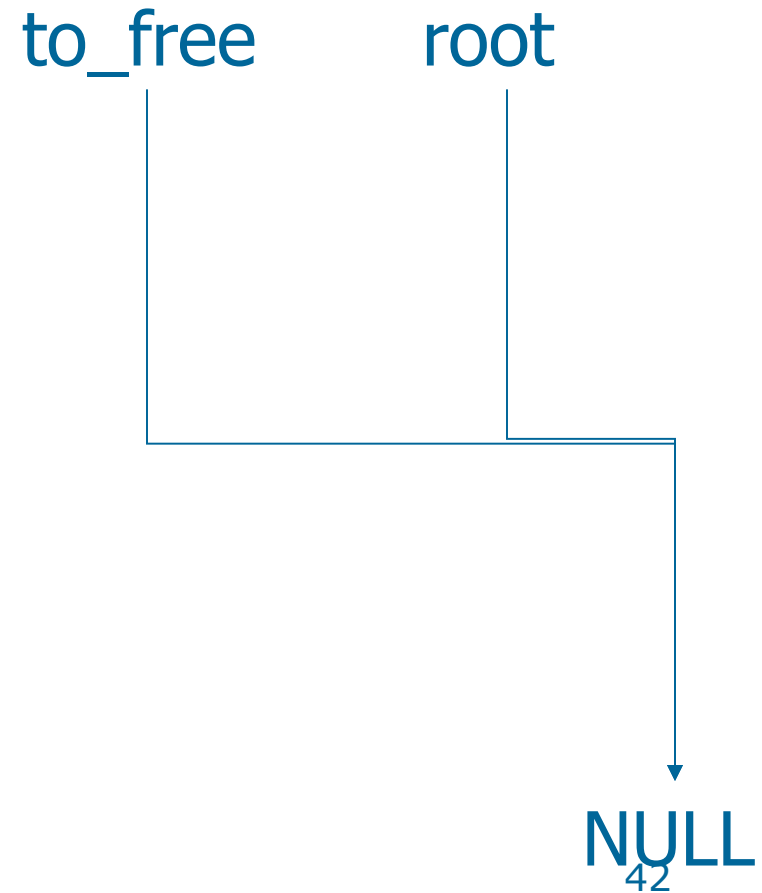
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



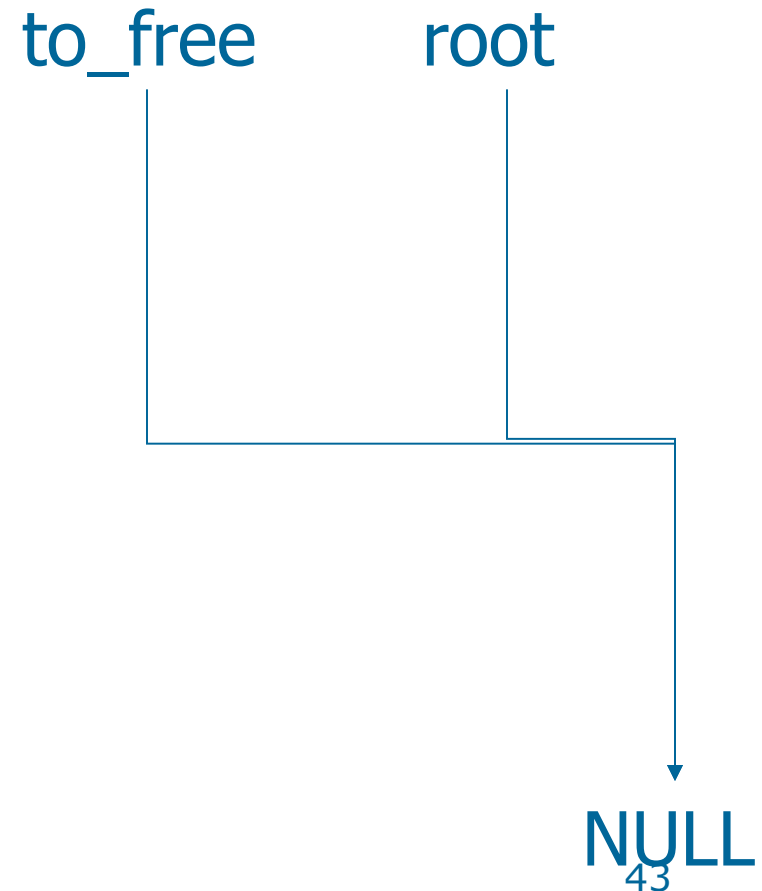
# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



# Freeing all nodes of a list

```
while (to_free != NULL)
{
    root = root->next;
    free(to_free);
    to_free = root;
}
```



# List reversing

- Given a list defined as:

```
struct list_int {  
    int val;  
    struct list_int *next;  
};  
...  
struct list_int *head=NULL;
```

- Write a function that reverse a list of this type.

# Exercise 3.5

- Add to program menu (Nokia DB List) function to reverse the list
- Check by Traverse function

# Tổng kết các chức năng của Bài tập

- 1. Import from NokiaDB.dat (insertafter)
- 2. Display (traverse)
- 3. Add new phone (insertbefore)
- 4. Insert at Position
- 5. Delete at Position
- 6. Delete current
- 7. Delete first
- 8. Reverse List
- 9. Save to File
- 10. Quit(Free)

# Exercise 3.6

- Develop a simple student management program using linked list composed of node like this:

```
typedef struct Student_t {
    char    id[ID_LENGTH];
    char    name[NAME_LENGTH];
    int     grade;

    struct Student_t *next;
} Student;
```

# Exercise 3.6 (cont)

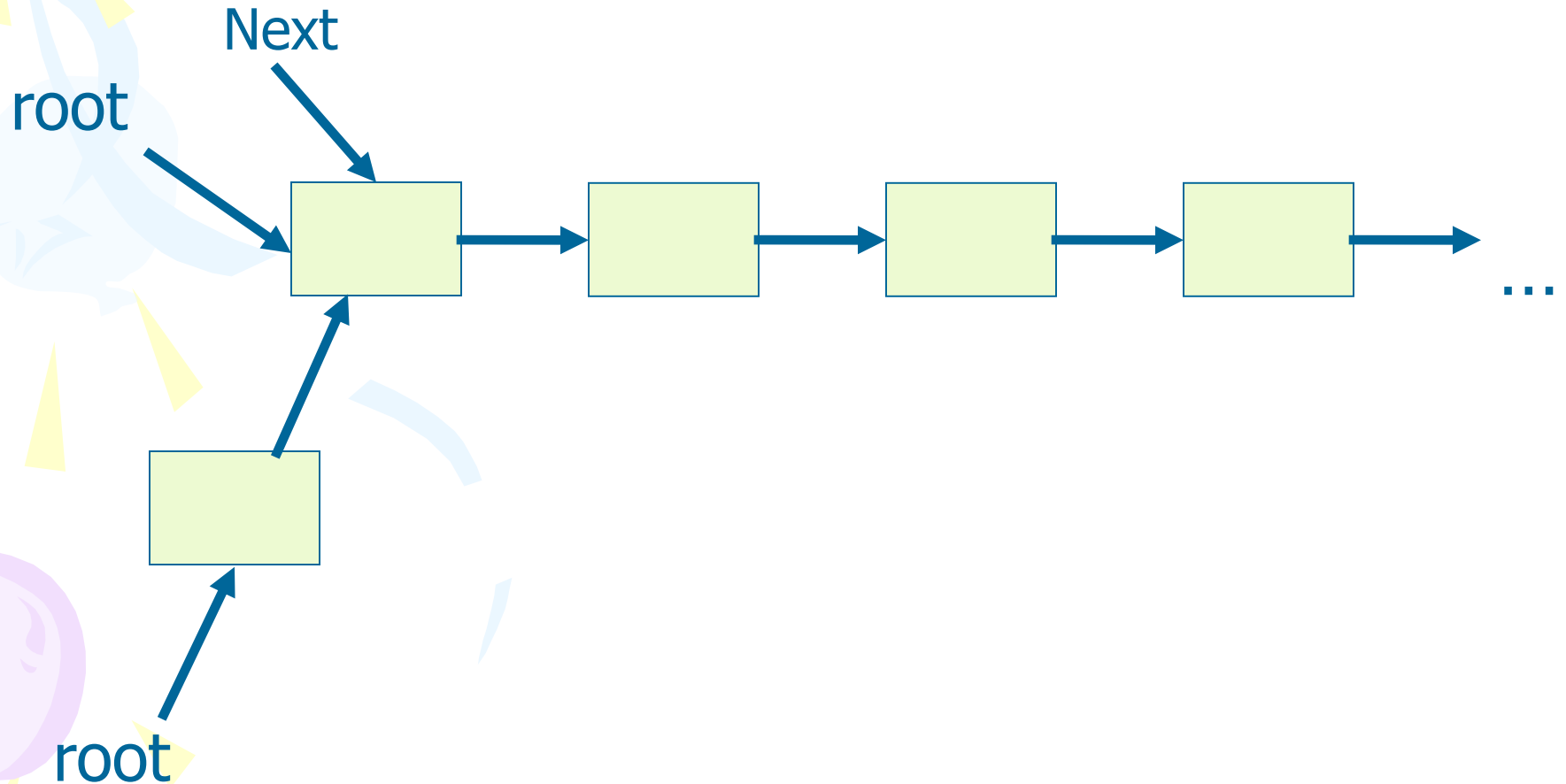
so that:

- The list is sorted in descending order of student's grades.
- Program provide the functionality of:
  - Insert new student (when you insert a new student into this list, first find the right position)
  - searching a student by ID: return to a pointer
  - delete a student with a given ID

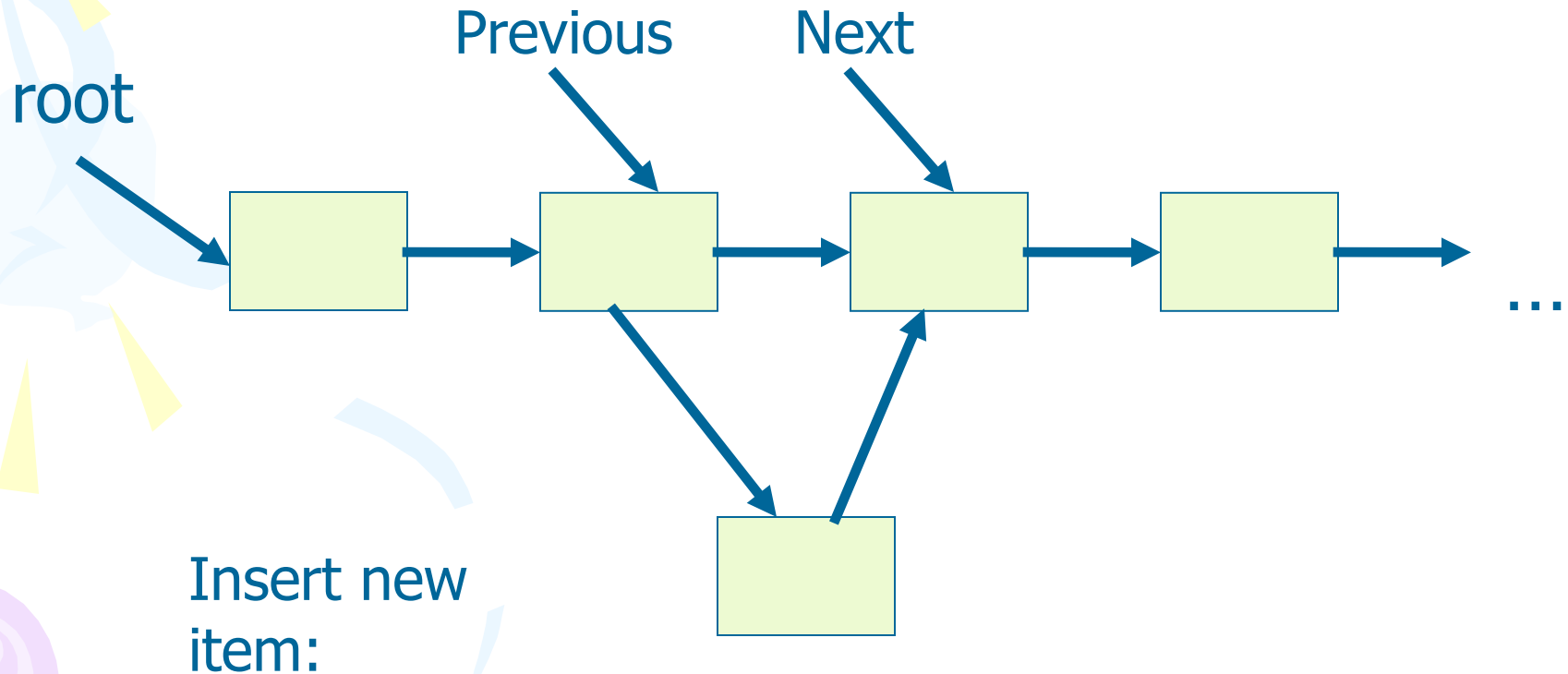
- ;



# Adding a student - beginning



# Adding a student – mid/end



```
Student *add_student(Student *root, Student *to_add)
```

```
{
```

```
    Student *curr_std, *prev_std = NULL;
```

```
    if (root == NULL) ← handle empty list  
        return to_add;
```

```
    if (to_add->grade > root->grade)  
    {  
        to_add->next = root;  
        return to_add;  
    }
```

← handle beginning

```
    curr_std = root;  
    while (curr_std != NULL && to_add->grade < curr_std->grade)  
    {  
        prev_std = curr_std;  
        curr_std = curr_std->next;  
    }
```

```
    prev_std->next = to_add;  
    to_add->next = curr_std;
```

```
    return root;
```

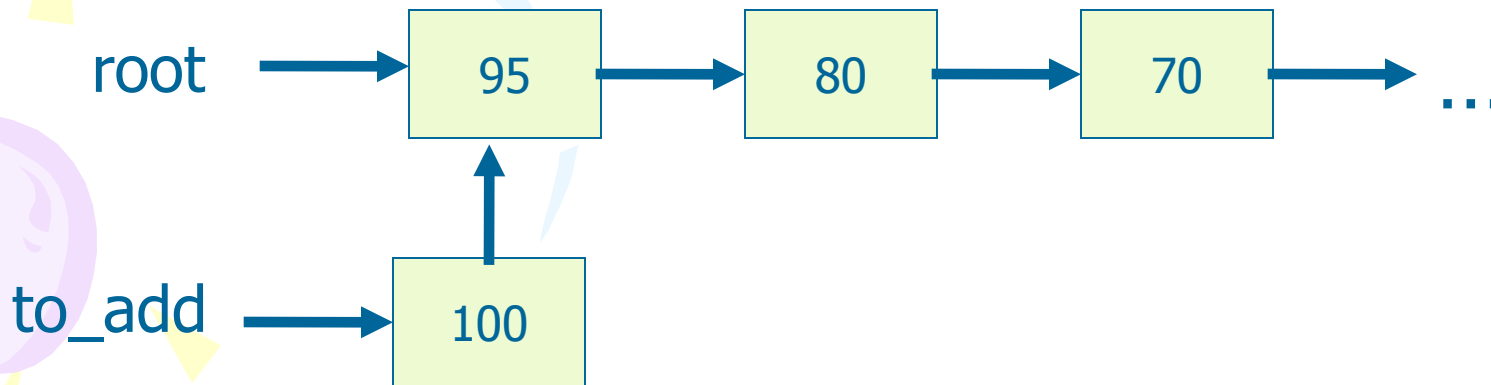
```
}
```

← the rest

# Adding a student – beginning

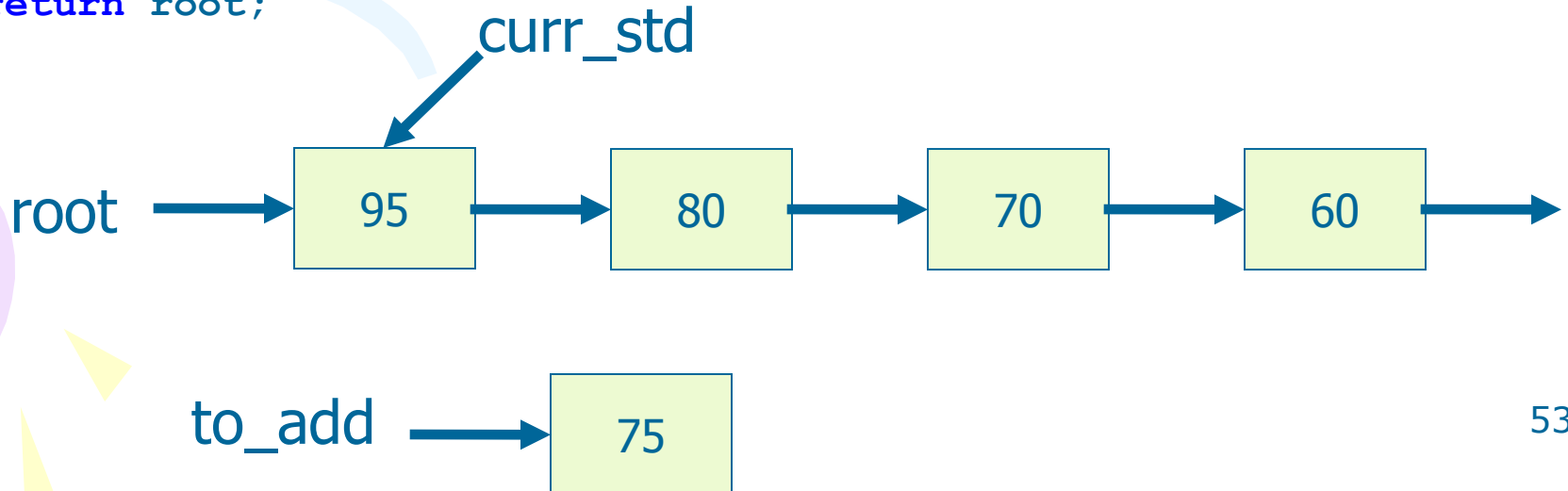
```
if (root == NULL)
    return to_add;

if (to_add->grade > root->grade)
{
    to_add->next = root;
    return to_add;
}
```



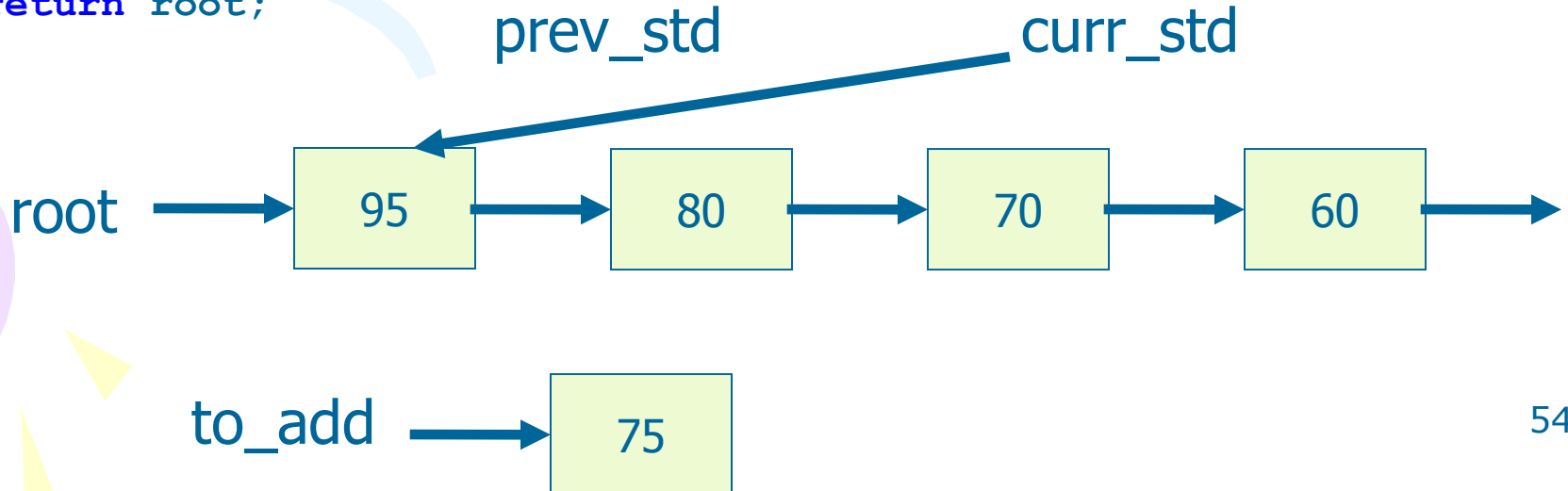
# Adding a student - mid / end

```
→ curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



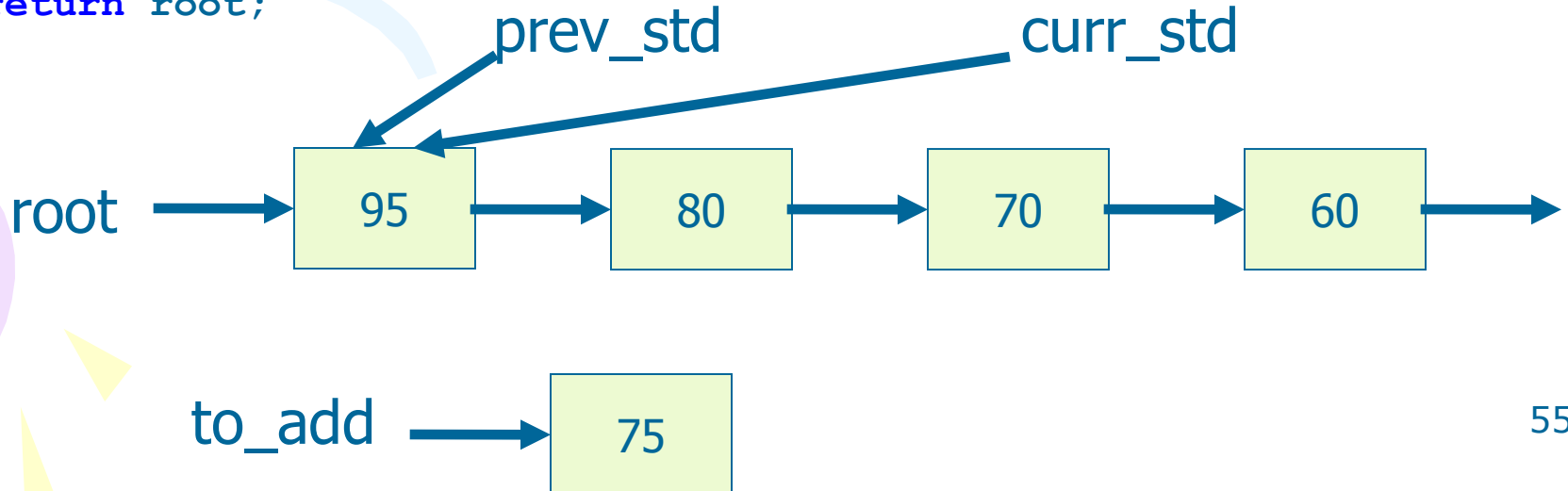
# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



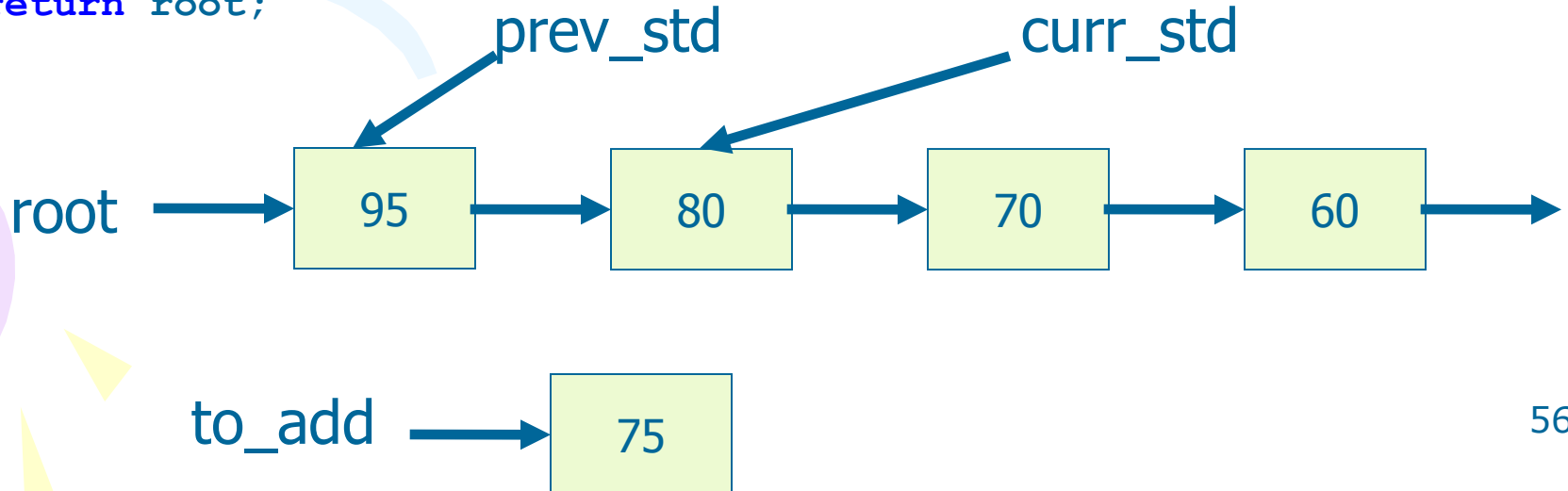
# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



# Adding a student - mid / end

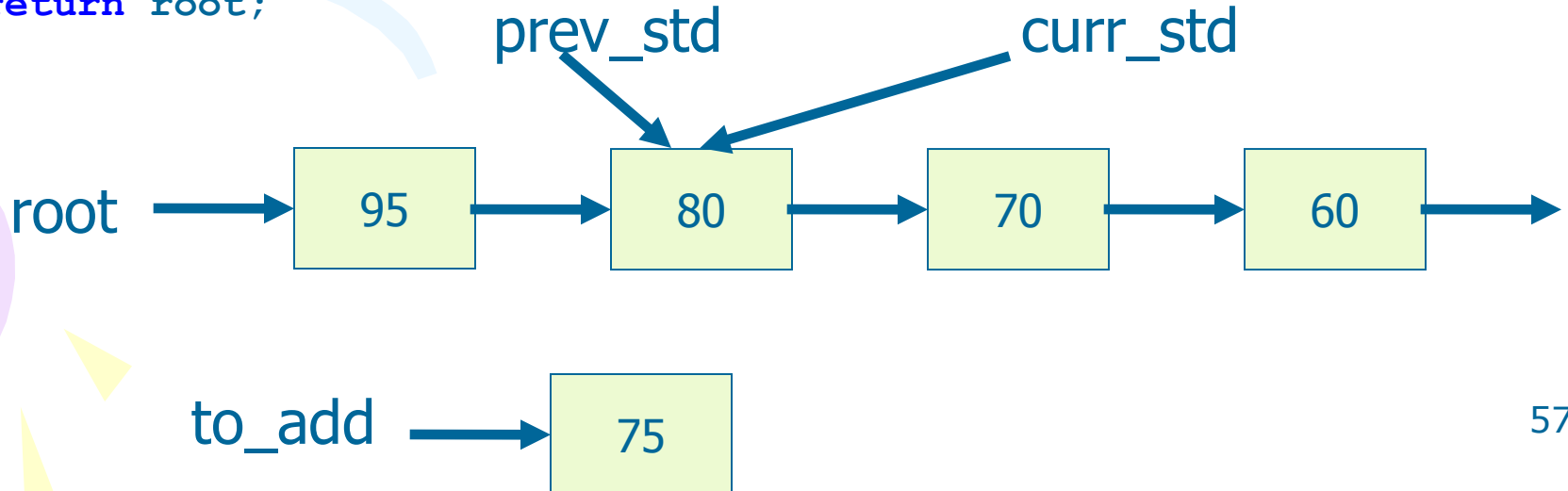
```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```





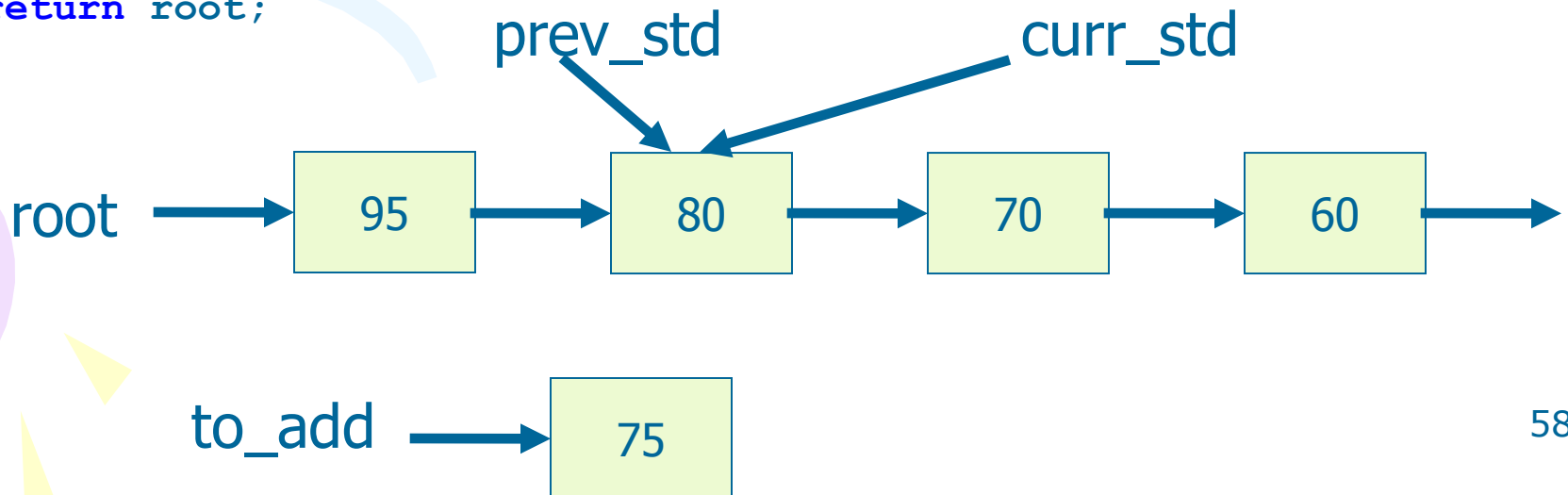
# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



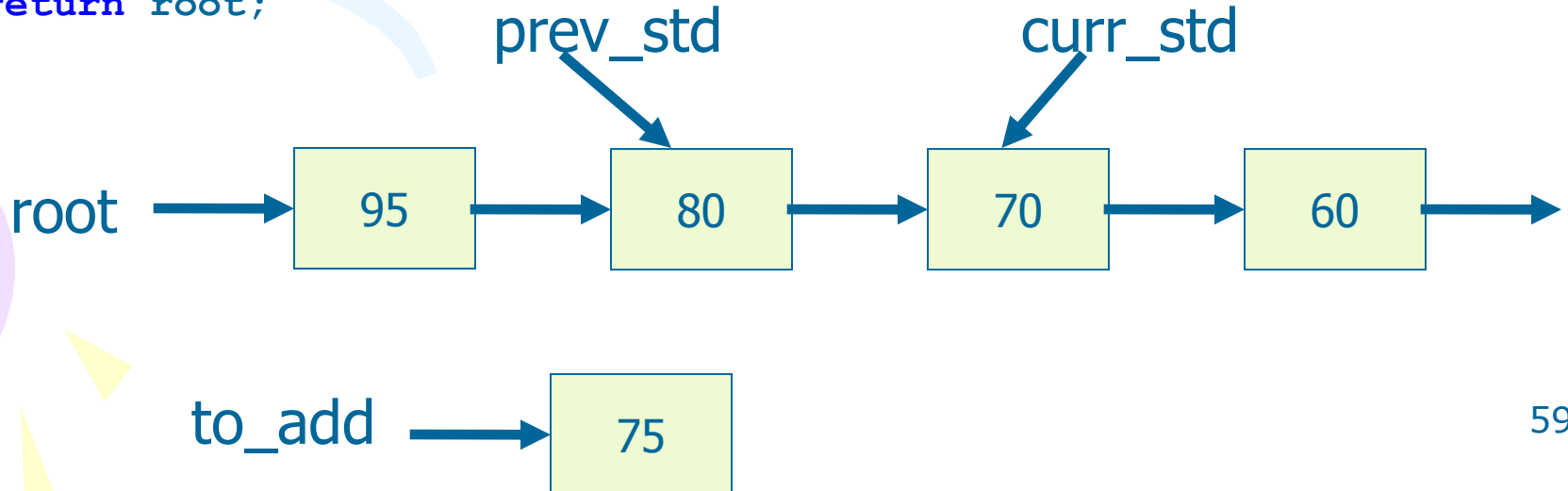
# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



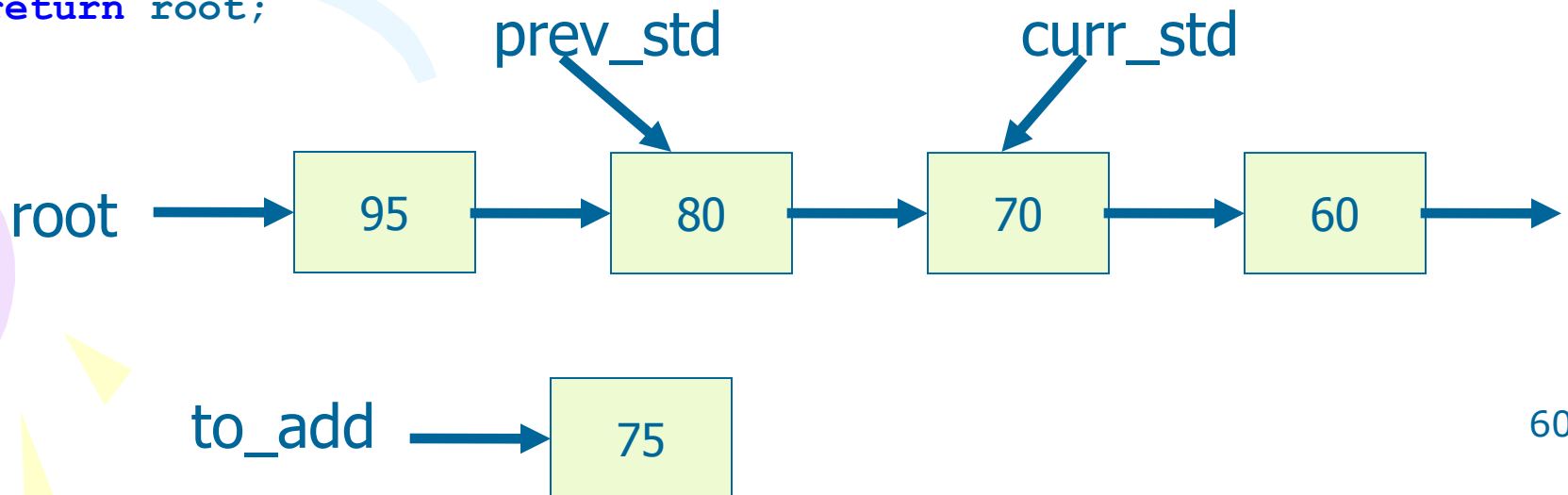
# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



# Adding a student - mid / end

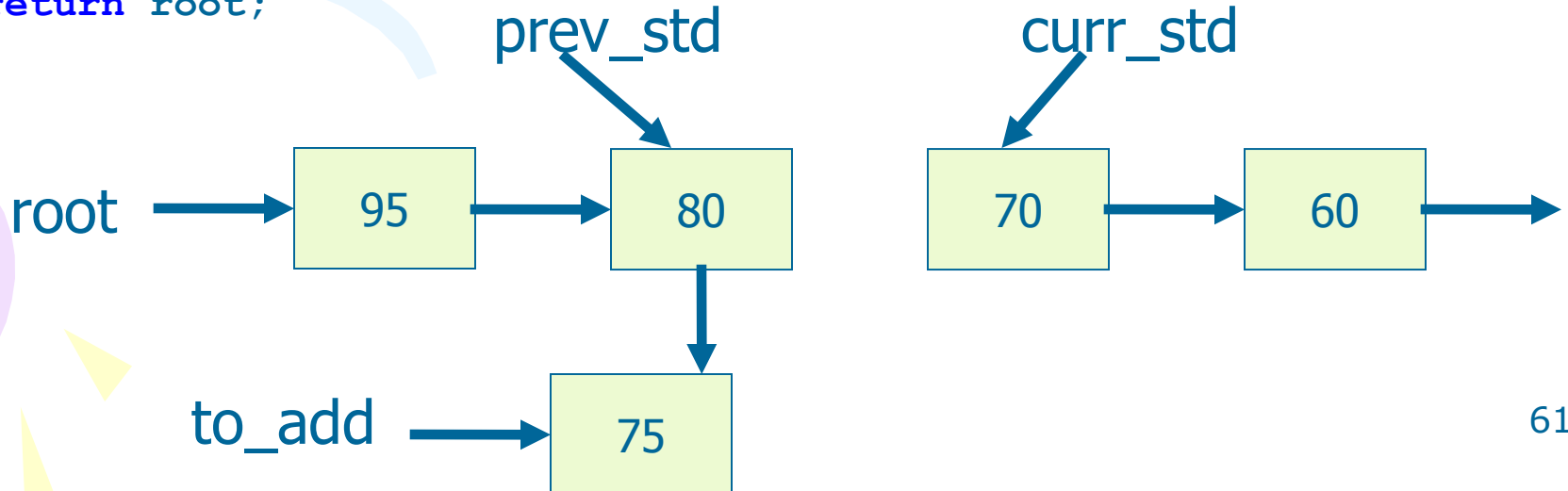
```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}  
  
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}
```

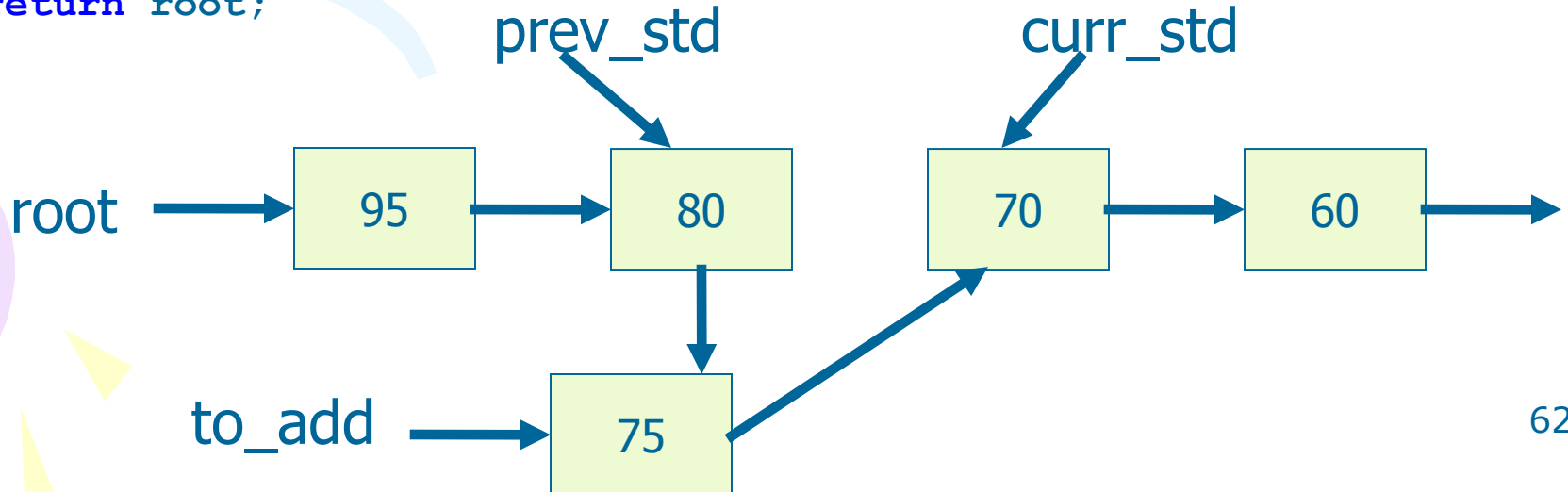
→  
prev\_std->next = to\_add;  
to\_add->next = curr\_std;  
return root;



# Adding a student - mid / end

```
curr_std = root;  
while (curr_std != NULL && to_add->grade < curr_std->grade)  
{  
    prev_std = curr_std;  
    curr_std = curr_std->next;  
}
```

```
prev_std->next = to_add;  
to_add->next = curr_std;  
return root;
```



# Exercise 3.7

- Implement `find_student`, which receives a list and an ID number and returns a pointer to a student whose ID matches or NULL if no such student is found.

```
Student *find_student(Student *root,  
                      char* id);
```

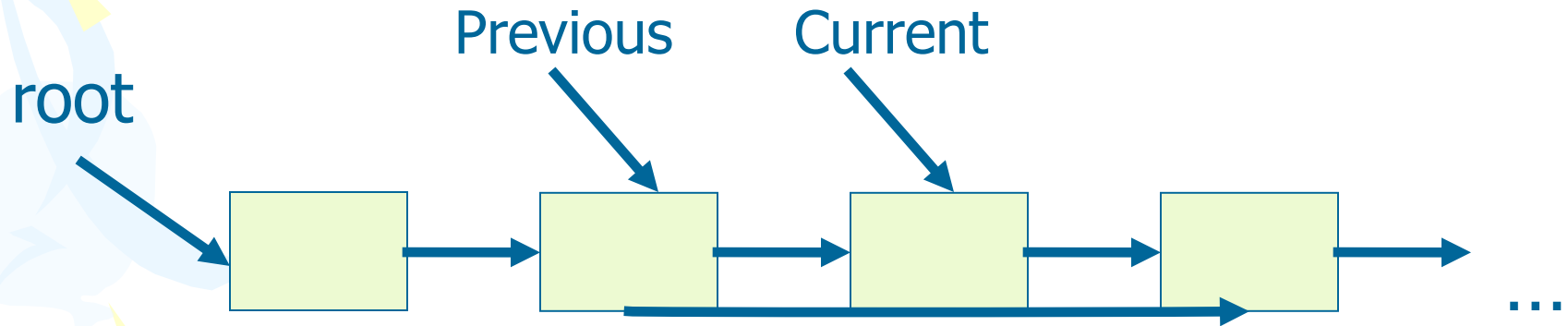
Hint: Use `strcmp(s1, s2)` which compares `s1` and `s2` and returns 0 if they are equal

# Removing a student

- We would like to be able to remove a student by her/his ID.
- The function that performs this is **remove\_student**



# Removing a student - reminder



# Removing a student - beginning

```
if (root == NULL)
    return root;
```

```
cur = root;
```

```
if (strcmp(cur->id, id) == 0)
{
    root = root->next;
    free(cur);
    return root;
}
```

ID

14525

cur

last

root

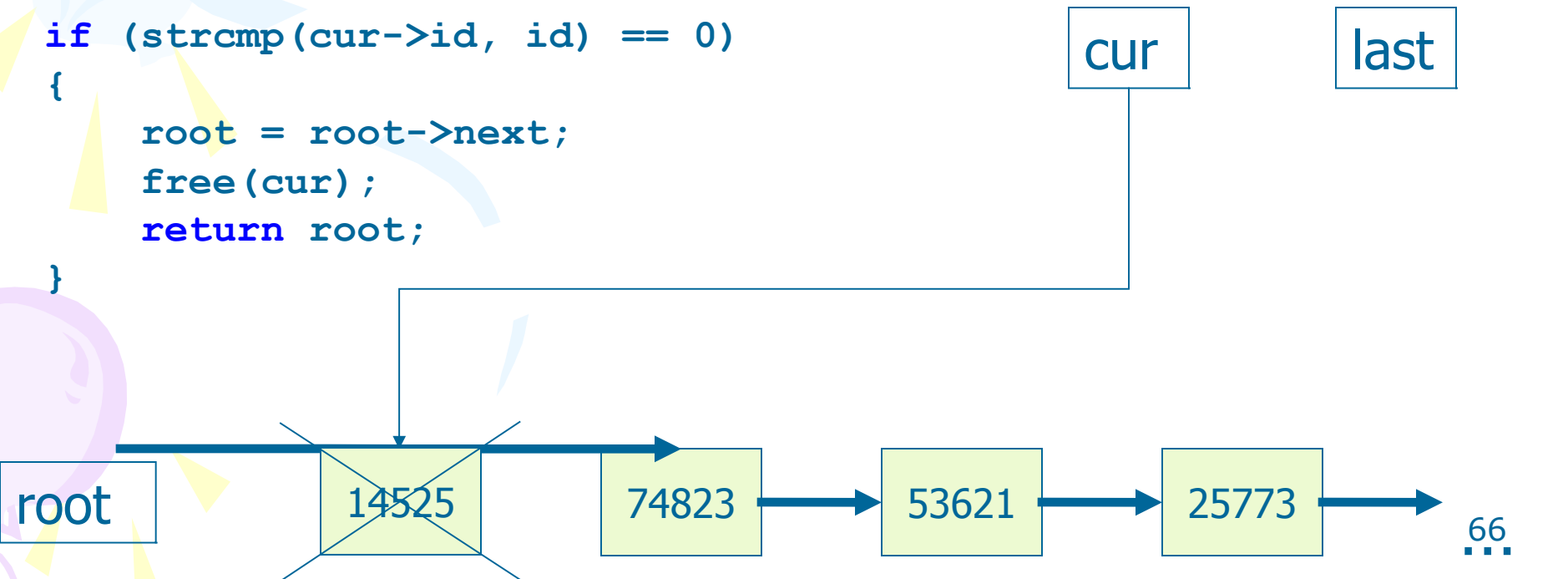
~~14525~~

74823

53621

25773

...



# Removing a student – mid list

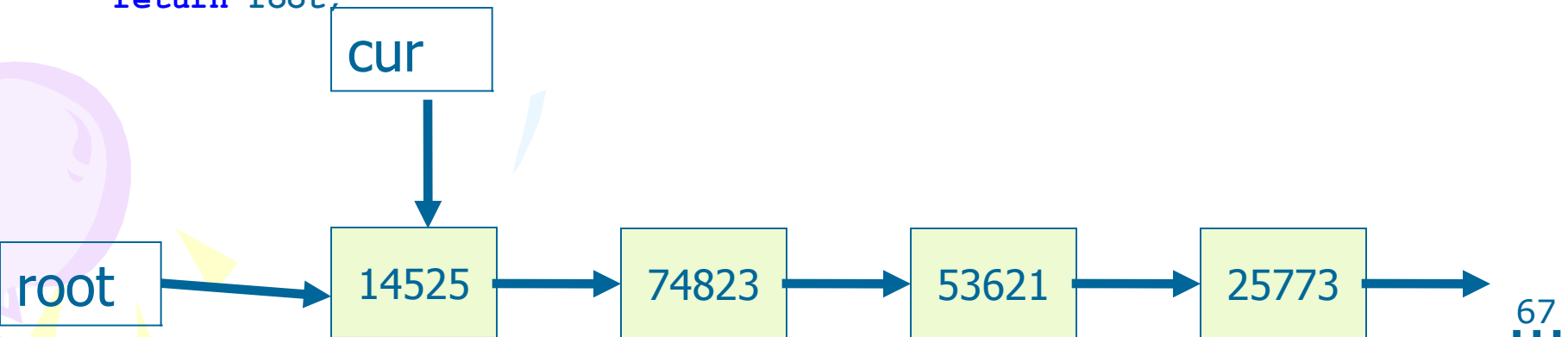
```
while (cur != NULL && strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

if (cur != NULL)
{
    prev->next = cur->next;
    free(cur);
}

return root;
```

ID

53621



# Removing a student – mid list

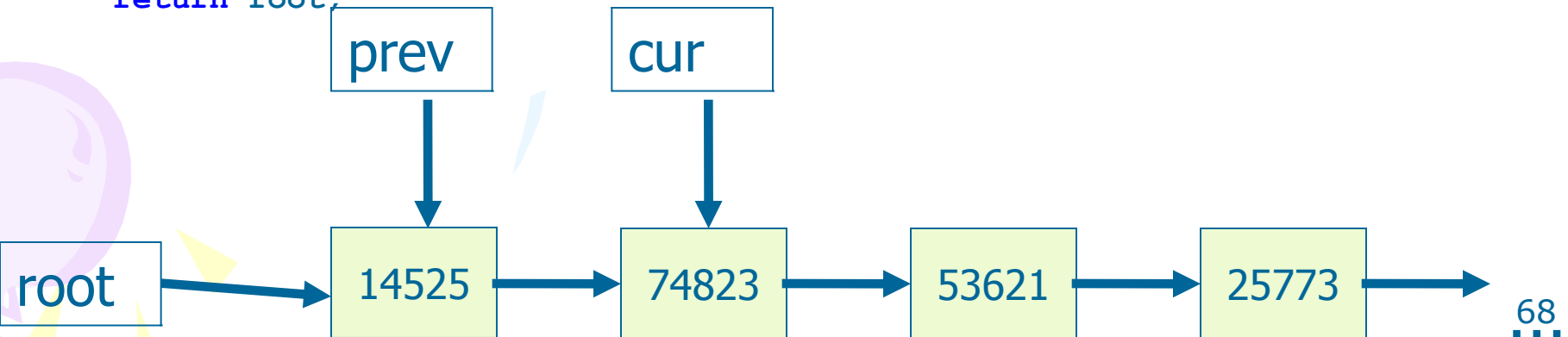
```
while (cur != NULL && strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

if (cur != NULL)
{
    prev->next = cur->next;
    free(cur);
}

return root;
```

ID

53621



# Removing a student – mid list

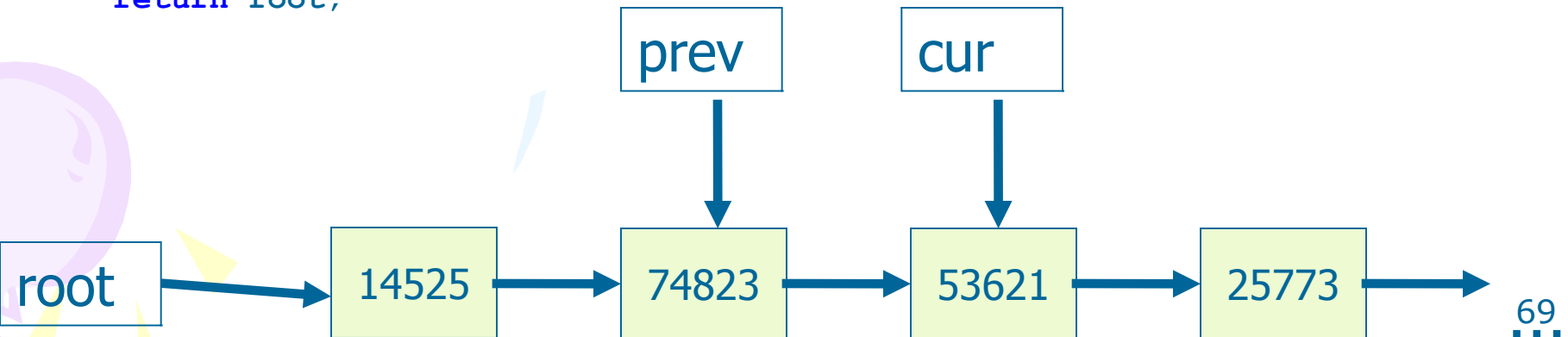
```
while (cur != NULL && strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

if (cur != NULL)
{
    prev->next = cur->next;
    free(cur);
}

return root;
```

ID

53621



# Removing a student – mid list

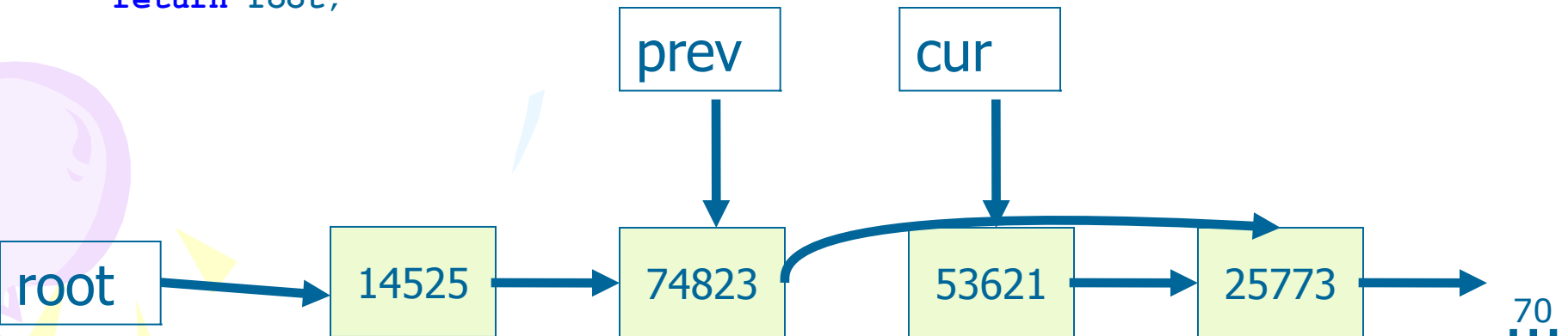
```
while (cur != NULL && strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

if (cur != NULL)
{
    prev->next = cur->next;
    free(cur);
}

return root;
```

ID

53621



# Removing a student – mid list

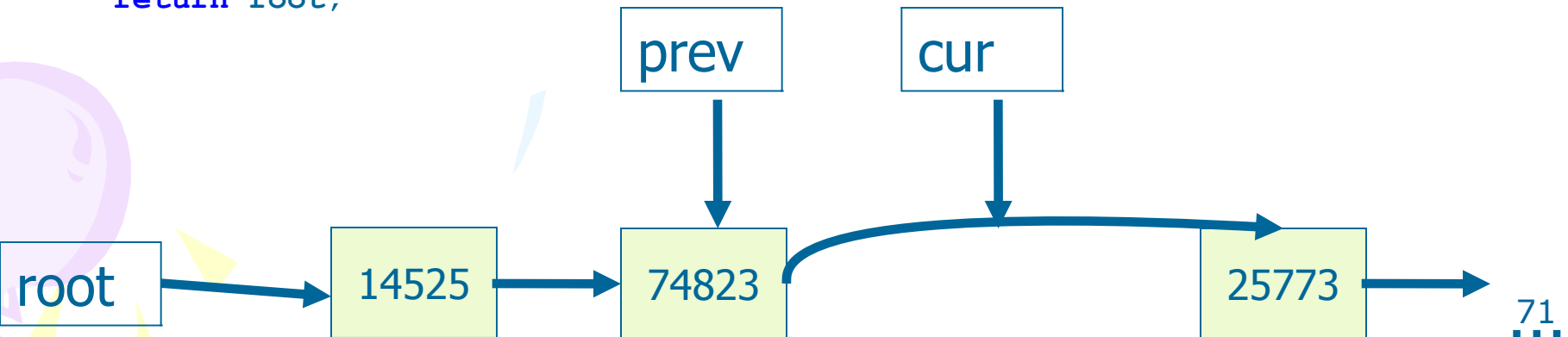
```
while (cur != NULL && strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

if (cur != NULL)
{
    prev->next = cur->next;
    free(cur);
}

return root;
```

ID

53621



# Exercise 3.8

- Add a **change\_grade** function. The function should take as parameters the root of the list, the ID whose grade we'd like to change, and the new grade
- Hint – Create a new student with the same name, ID as the old one, with the new grade. Then remove the old student from the list and add the new one using the existing functions



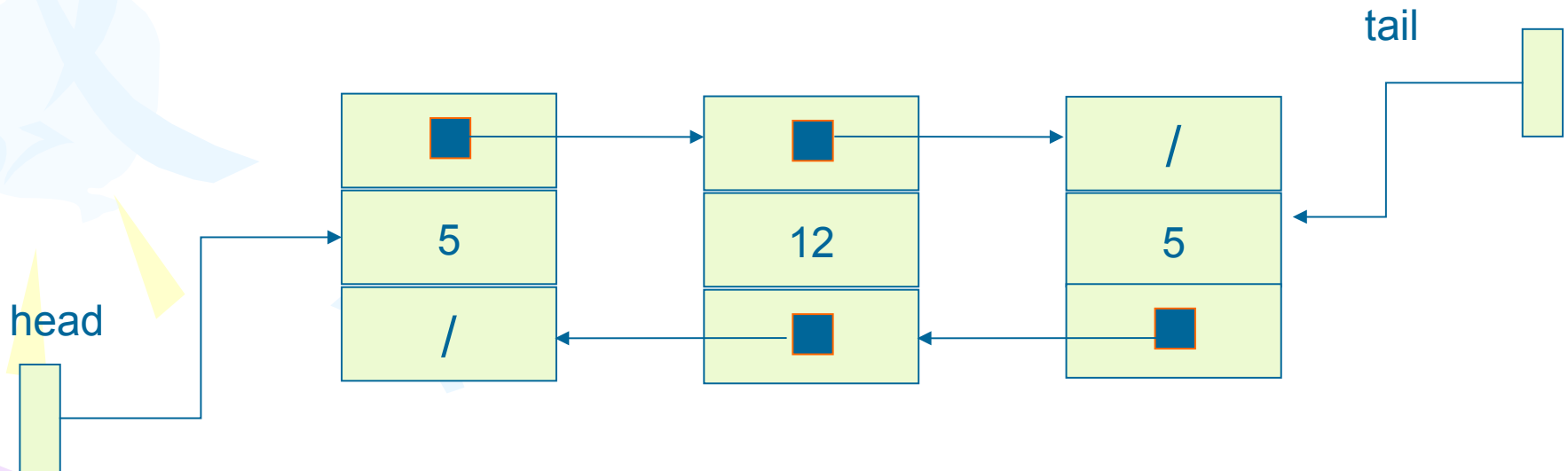
# Homework 1

- We are now designing “address list” for mobile phones.
- You must declare a record structure that can keep a name, a phone number, and a e-mail address at least. And you must make the program which can deals with any number of the data.
- Hint: you can organize elements and data structure using following record structure AddressList

```
struct AddressList {  
    struct AddressList *prev;  
    struct AddressList *next;  
    struct Address addr;  
};
```

# Double link list

- An element has 2 pointer fields, we can follow front and back.



# Declaration

```
typedef ... ElementType;
```

```
typedef struct Node{  
    ElementType Element;  
    Node* Prev;  
    Node* Next;  
};
```

```
typedef Node* Position;
```

```
typedef Position DoubleList;
```

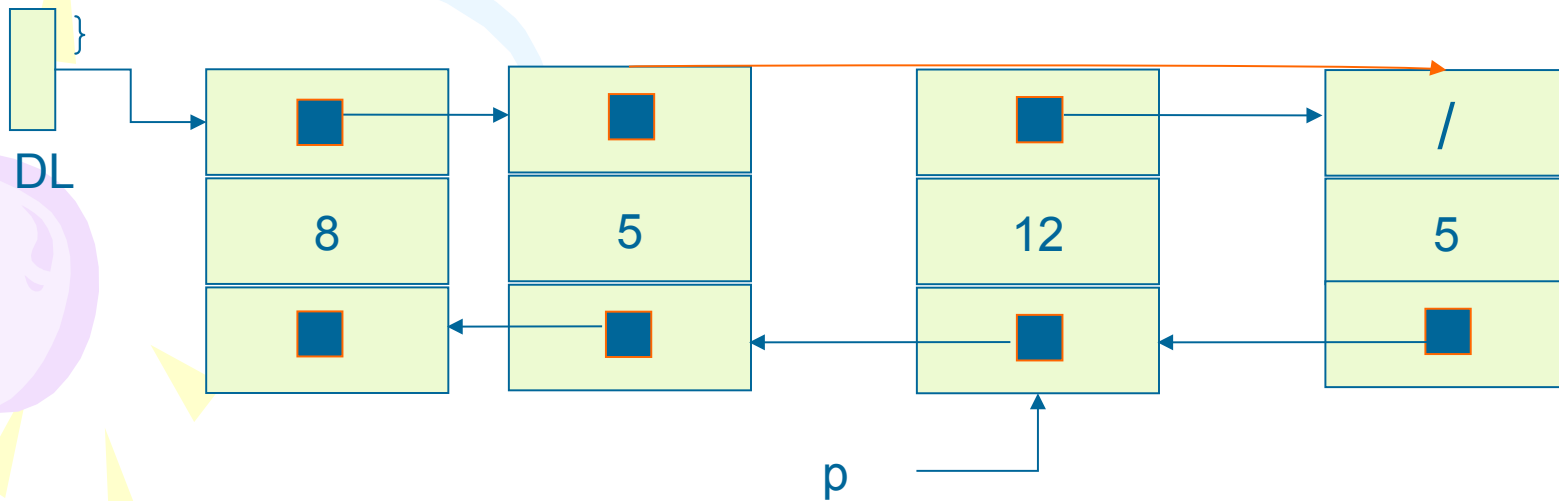
# Initialisation and check for emptiness

```
void MakeNull_List (DoubleList *DL) {  
    (*DL) = NULL;  
}
```

```
int Empty (DoubleList DL) {  
    return (DL==NULL);  
}
```

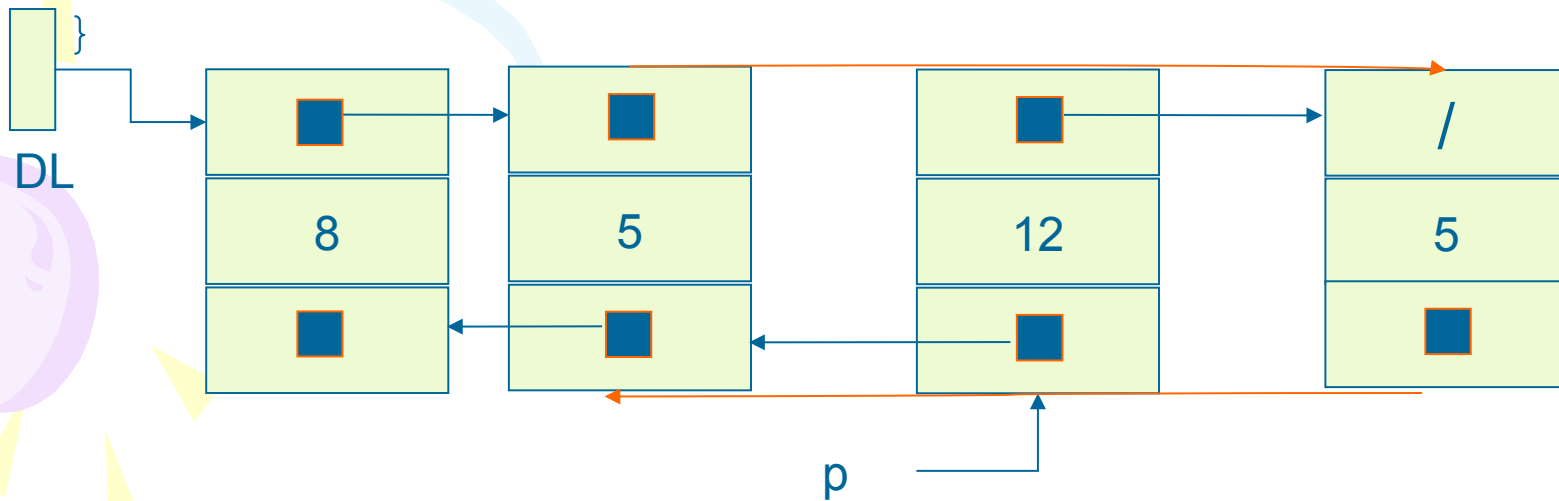
# Delete a node pointed by p

```
void Delete_List (Position p, DoubleList *DL) {  
    if (*DL == NULL) printf("Empty list");  
    else {  
        if (p==*DL) (*DL)=(*DL)->Next;  
        //Delete first element  
        else p->Previous->Next=p->Next;  
        if (p->Next!=NULL) p->Next->Previous=p->Previous;  
        free(p);  
    }  
}
```



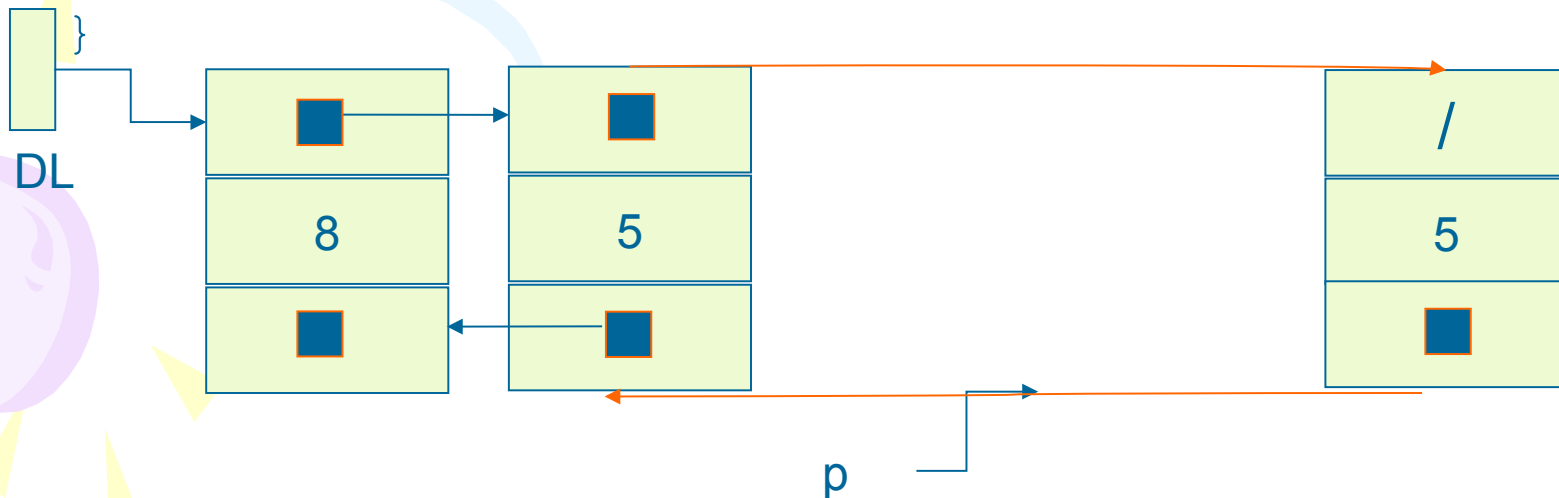
# Delete a node pointed by p

```
void Delete_List (Position p, DoubleList *DL) {  
    if (*DL == NULL) printf("Empty list");  
    else {  
        if (p==*DL) (*DL)=(*DL)->Next;  
        //Delete first element  
        else p->Previous->Next=p->Next;  
        if (p->Next!=NULL) p->Next->Previous=p->Previous;  
        free (p) ;  
    }  
}
```



# Delete a node pointed by p

```
void Delete_List (Position p, DoubleList *DL) {  
    if (*DL == NULL) printf("Empty list");  
    else {  
        if (p==*DL) (*DL)=(*DL)->Next;  
        //Delete first element  
        else p->Previous->Next=p->Next;  
        if (p->Next!=NULL) p->Next->Previous=p->Previous;  
        free (p);  
    }  
}
```



# Insertion

```
void Insert_List (ElementType X, Position p, DoubleList *DL){
    if (*DL == NULL){ // List is empty
        (*DL)=(Node*)malloc(sizeof(Node));
        (*DL)->Element = X;
        (*DL)->Previous = NULL;
        (*DL)->Next = NULL;
    }
    else{
        Position temp;
        temp=(Node*)malloc(sizeof(Node));
        temp->Element=X;
        temp->Next=p;
        temp->Previous=p->Previous;
        if (p->Previous!=NULL)
            p->Previous->Next=temp;
        p->Previous=temp;
    }
}
```



# Create a library

- Create lib.h
  - Type declaration
  - Function prototype
- Create lib.c
  - #include "lib.h"
  - Function Implementation
- Main Program: pro.c
  - #include "lib.h"

- Compile
- gcc -o ex pro.c lib.c

Another way:

```
gcc -c lib.c
gcc -c pro.c
gcc -o lib.o pro.o
ex
```

# Homework 2

- Using the library doubly linked list and the data you have created in the PhoneBook Exercise. Write a function that:
  - Count the max number of identical phone number elements in a list
  - Split these elements from the list, display the memory address of these element before and after the split action