Programming

Linked Lists

Linked Lists

- Dynamic
- Efficient use of memory
  - Allocate just as much as needed
- Easy insertion in front
- Local deletion

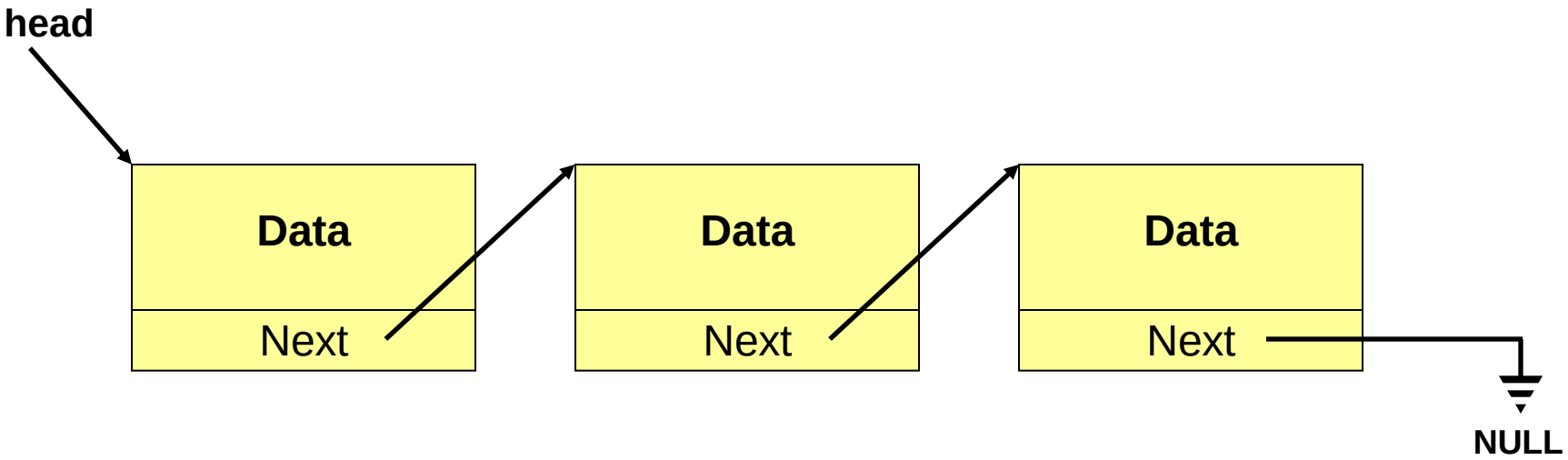
But…

- Hard to get to any particular element

Linked Lists

- A list is a chain of nodes.

```
typedef struct node_type
{
    <data>
    struct node_type* next;
} Node;
```



Linked Lists Operations

- Insert
  - front, back, middle
- Delete
- Find
- Size

Course Management System

- Maintain a list of students
  - Keep their ID, name, grade etc.
- Allow for adding / removing a student
- Find a student in the list
- Produce Reports
  - For example: Average grade

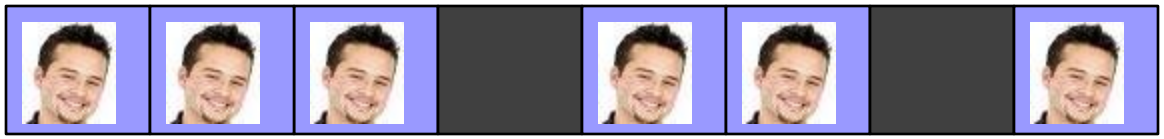
Storing a Collection of Students

Suggestion 1

Use an array of student structures

- Problems:
  - What size is the array?
  - Remove students – either expensive or creates "holes"
  - Insertion in a sorted array is expensive

"holes":



Linking Students

Suggestion 2

Use a **linked list**

Define a student node:

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

     /* A pointer to the next node in the list */
     struct student *next;
} sStudent;
```

Exercise

- Download find_student_ex.c from the tirgul home page
- Implement:

```
sStudent* find_student(const sStudent *head, const char* id)
```

- **`find_student`** searches for a student with a given id. It returns a pointer to the student if found, otherwise it returns NULL.

Creating a New Student

```
sStudent* new_student(char* name, char* id,
                      int grade)
{
    sStudent *std =(sStudent*)malloc(sizeof(sStudent));

    if (std != NULL)
    {
        strcpy(std->name, name);
        strcpy(std->id, id);
        std->grade = grade;
        std->next = NULL;
    }
    return std;
}
```

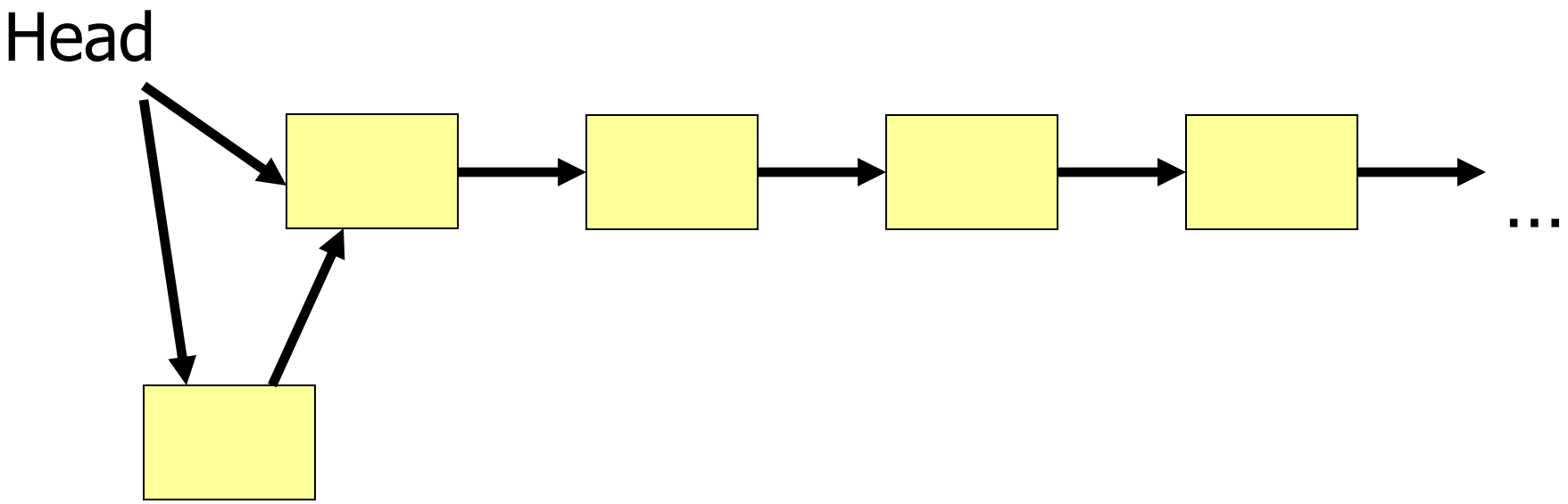
Add in Front

```
sStudent* add_front(sStudent *head, sStudent *std)
{
    std->next = head;
    return std;
}

int main()
{
    sStudent *std_list, *std;
    ...
    std = new_student(...);
    if (std != NULL )
        std_list = add_front(std_list, std);

    ...
    return 0;
}
```

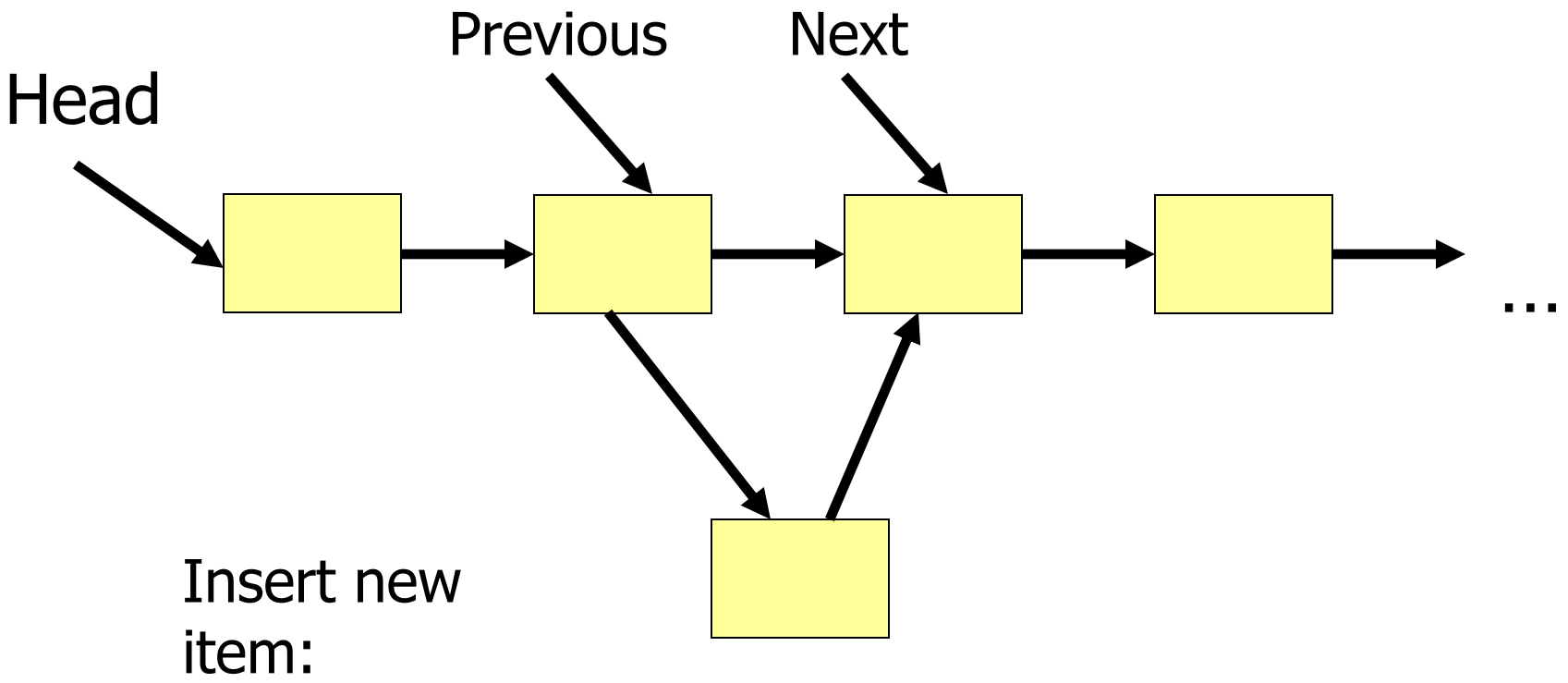
Sorted Add

- If the list is sorted (for example by grade), a student insertion should keep the list sorted
- We will implement this in a separate function

Adding a student - begining



Adding a student – mid/end



```
sStudent *add_student(sStudent *head, sStudent *to_add)
{
    sStudent *curr_std, *prev_std = NULL;

    if (head == NULL)
        return to_add;

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

    curr_std = head;
    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 head;
}
```

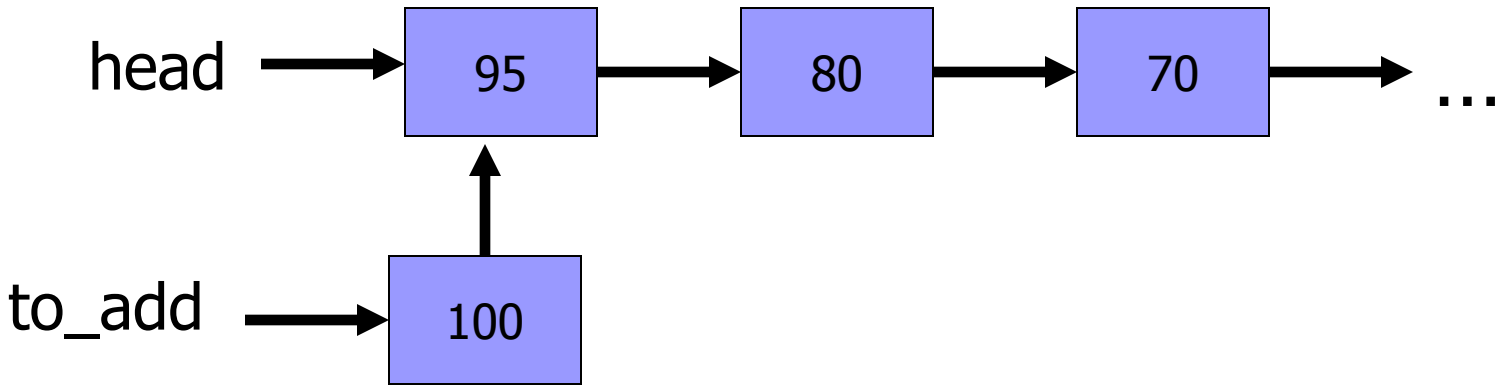
handle empty list

handle beginning

the rest

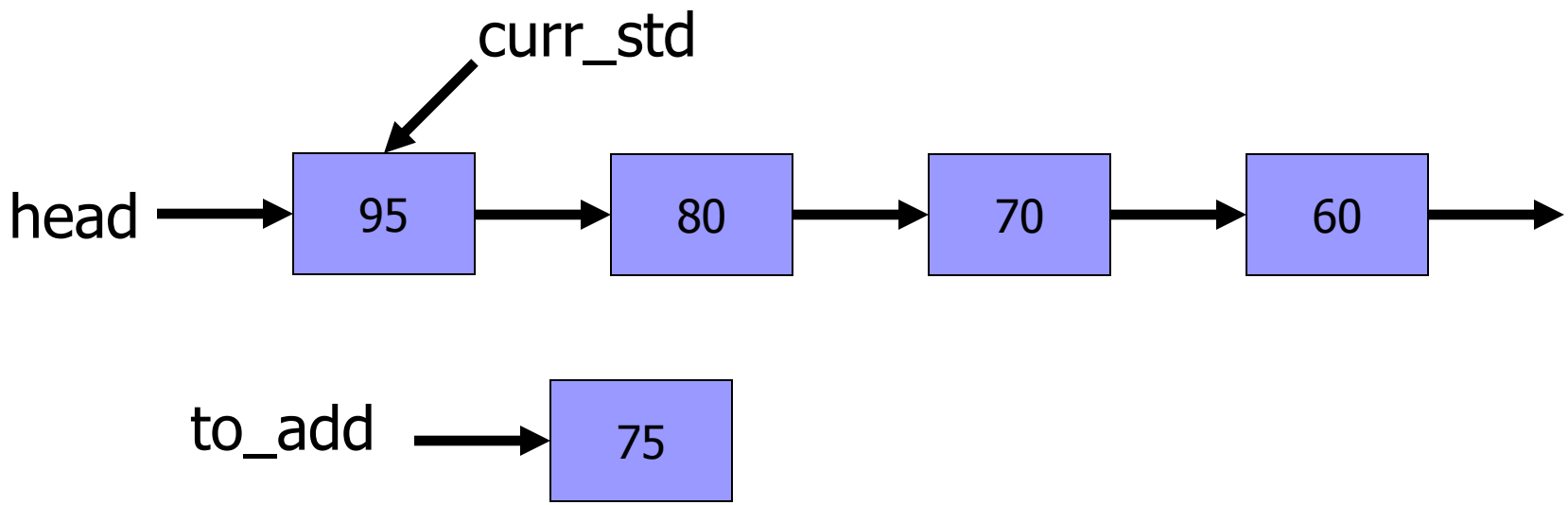
Adding a student – beginning

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

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



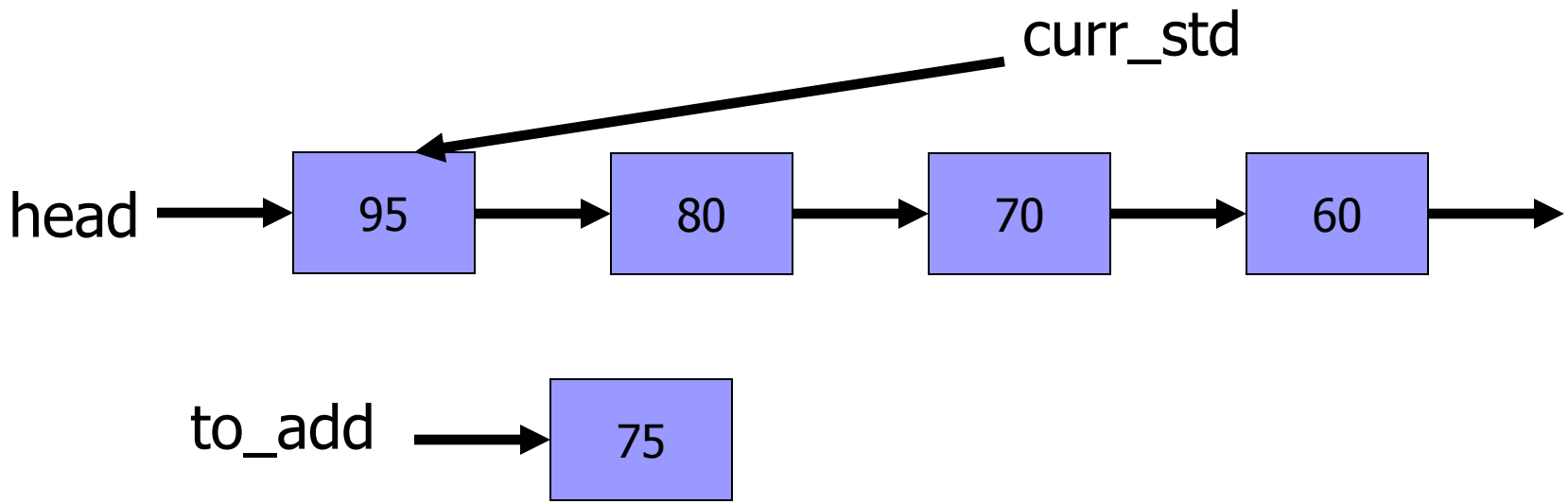
Adding a student – mid / end

```
curr_std = head;
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 head;
```



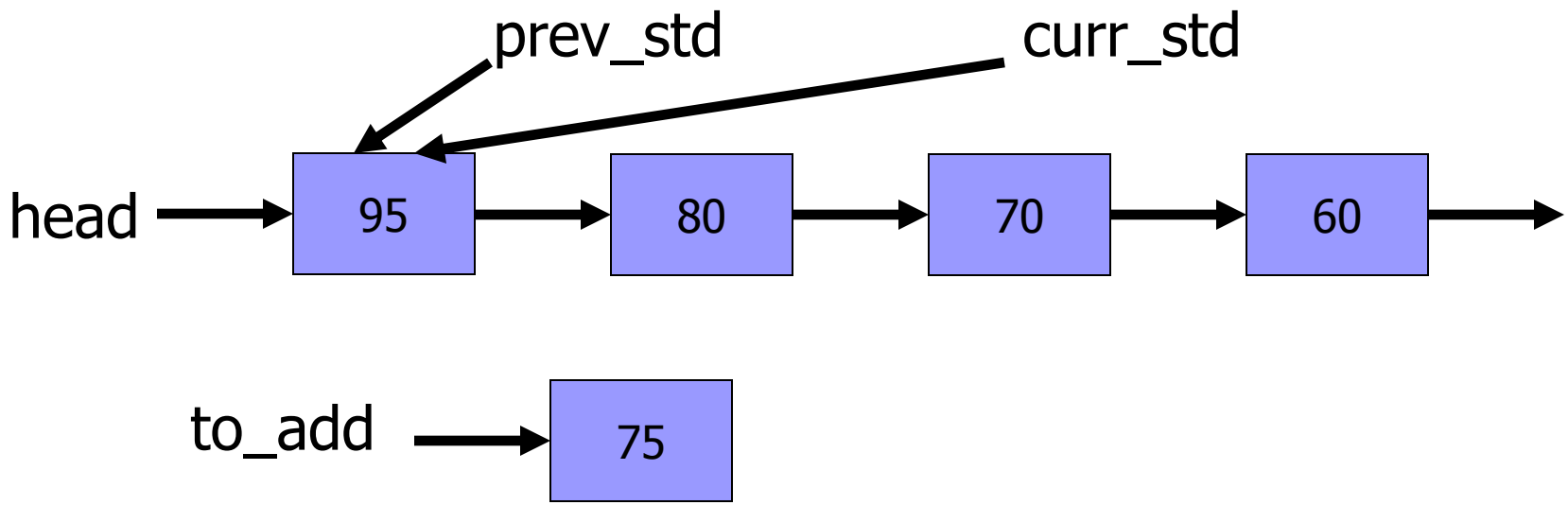
Adding a student – mid / end

```
curr_std = head;
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 head;
```



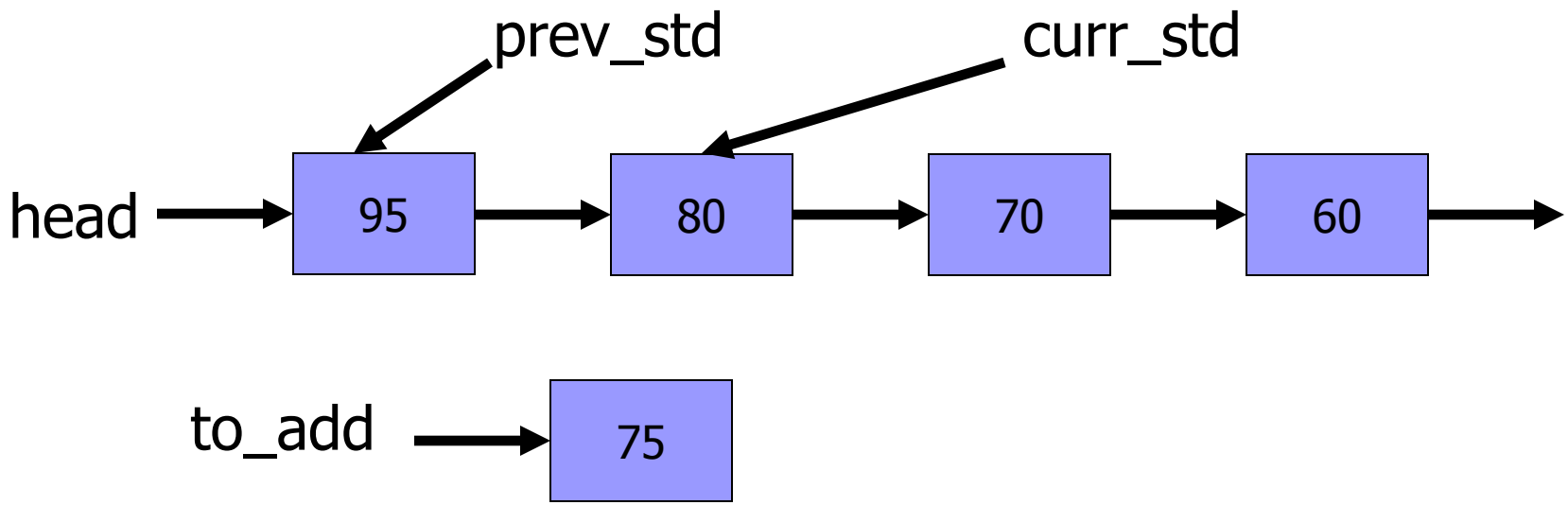
Adding a student – mid / end

```
curr_std = head;
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 head;
```



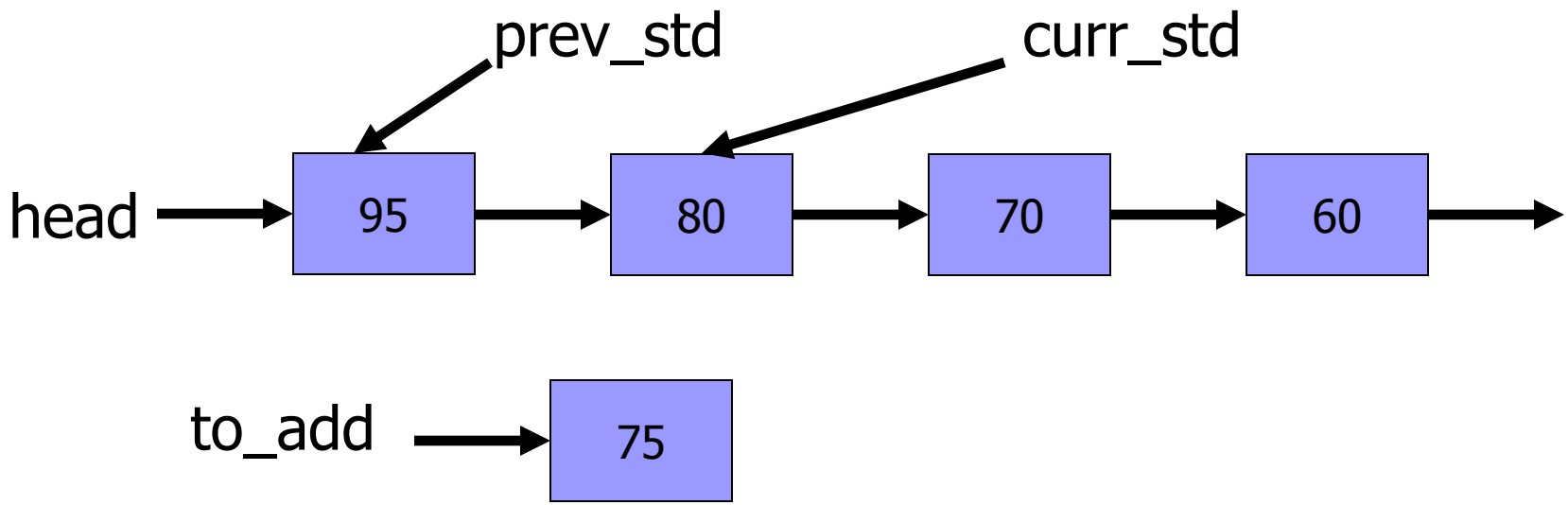
Adding a student – mid / end

```
curr_std = head;
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 head;
```



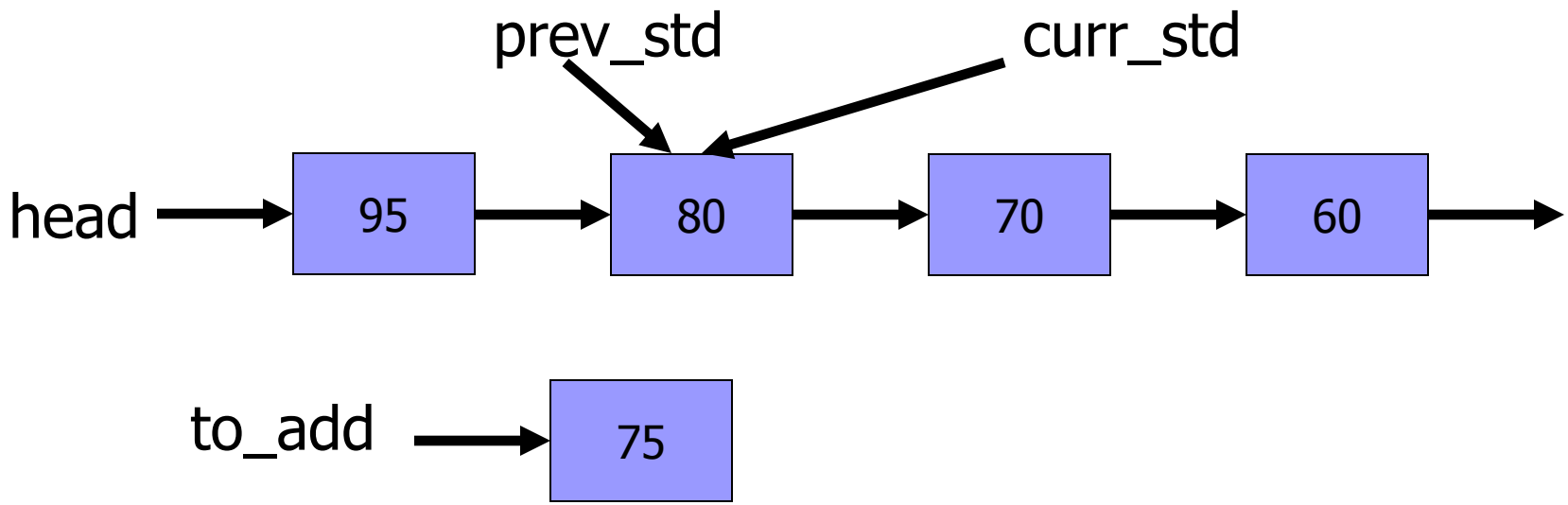
Adding a student – mid / end

```
curr_std = head;
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 head;
```



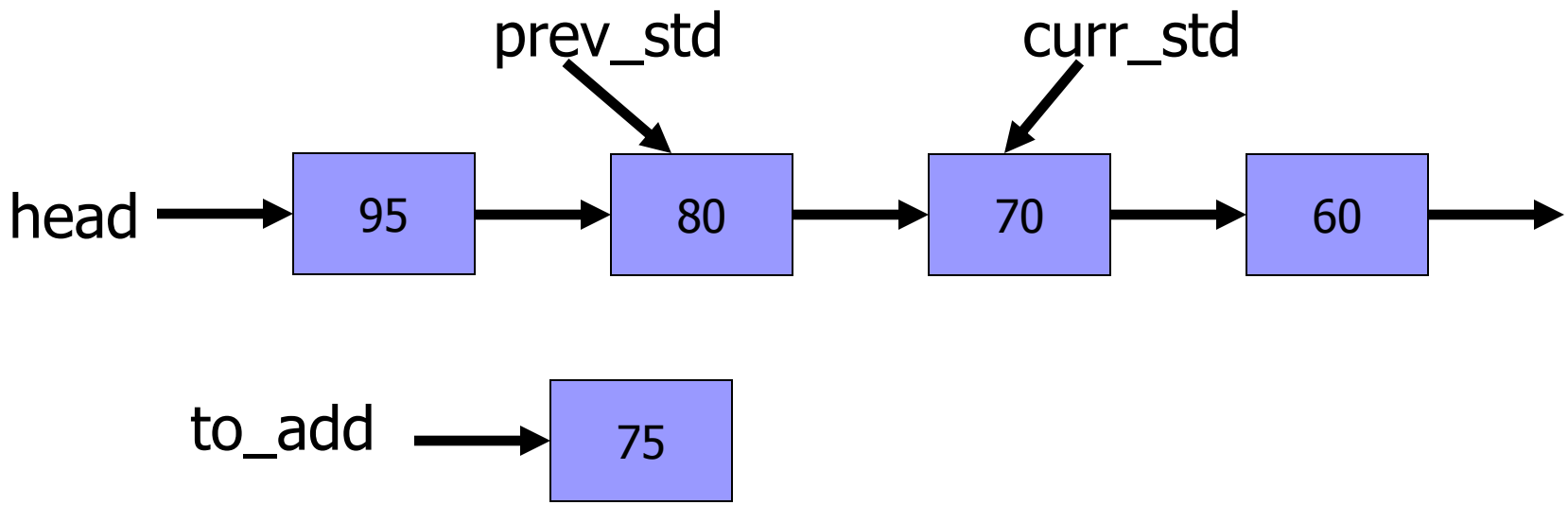
Adding a student – mid / end

```
curr_std = head;
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 head;
```



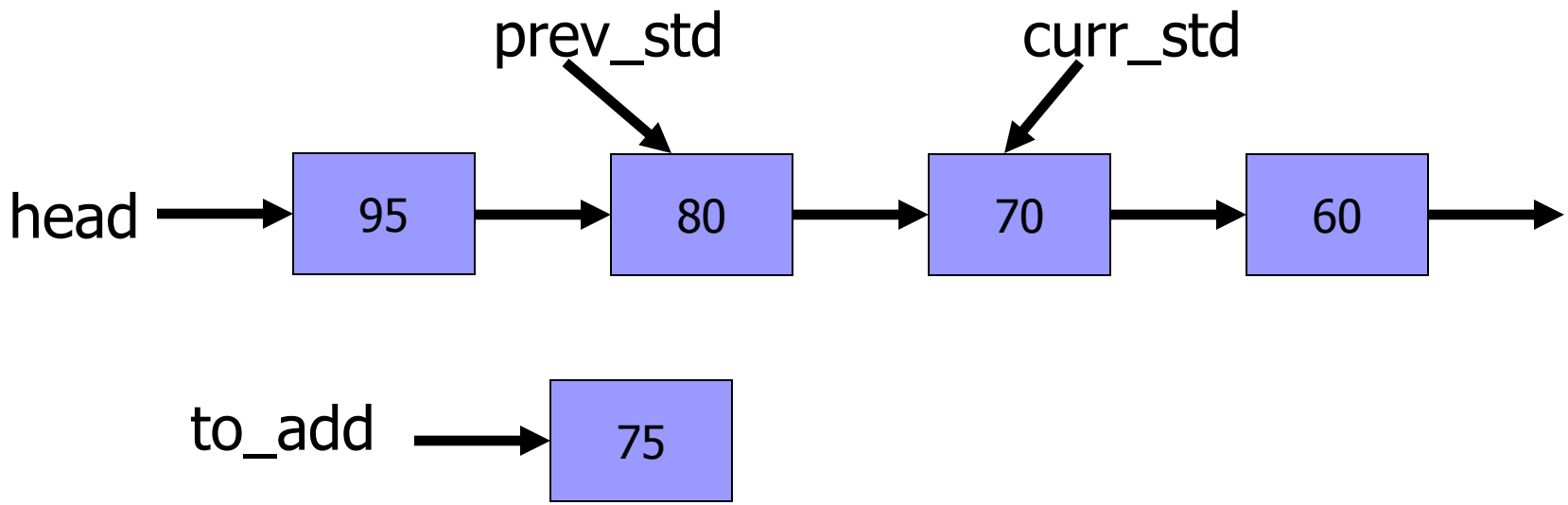
Adding a student – mid / end

```
curr_std = head;
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 head;
```



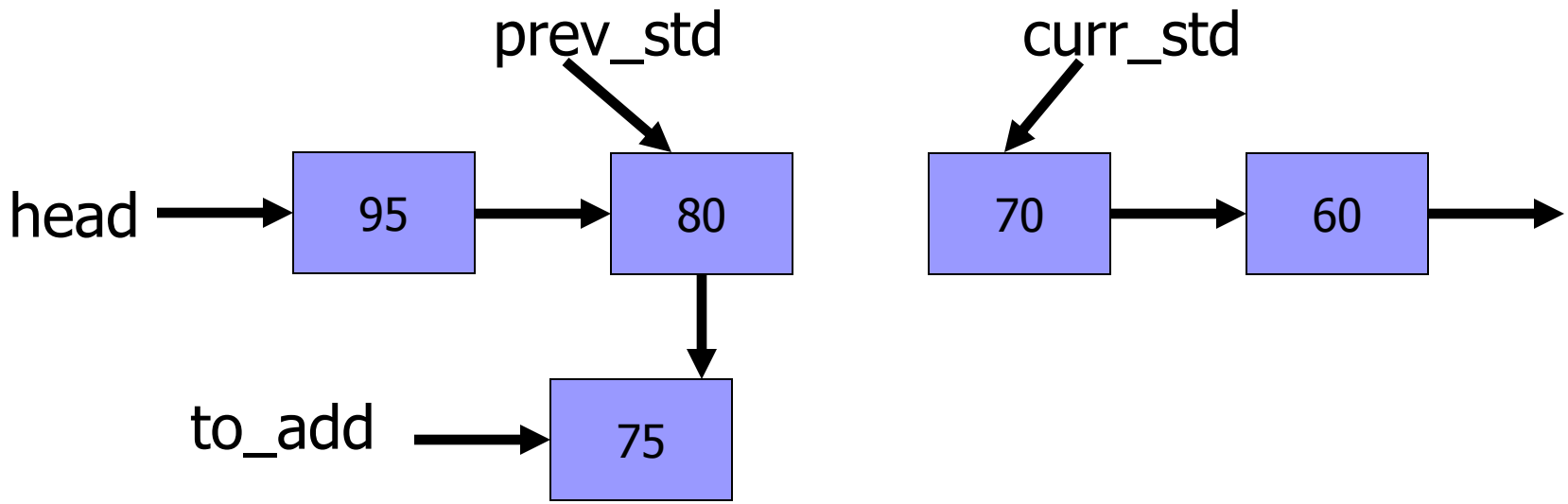
Adding a student – mid / end

```
curr_std = head;
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 head;
```



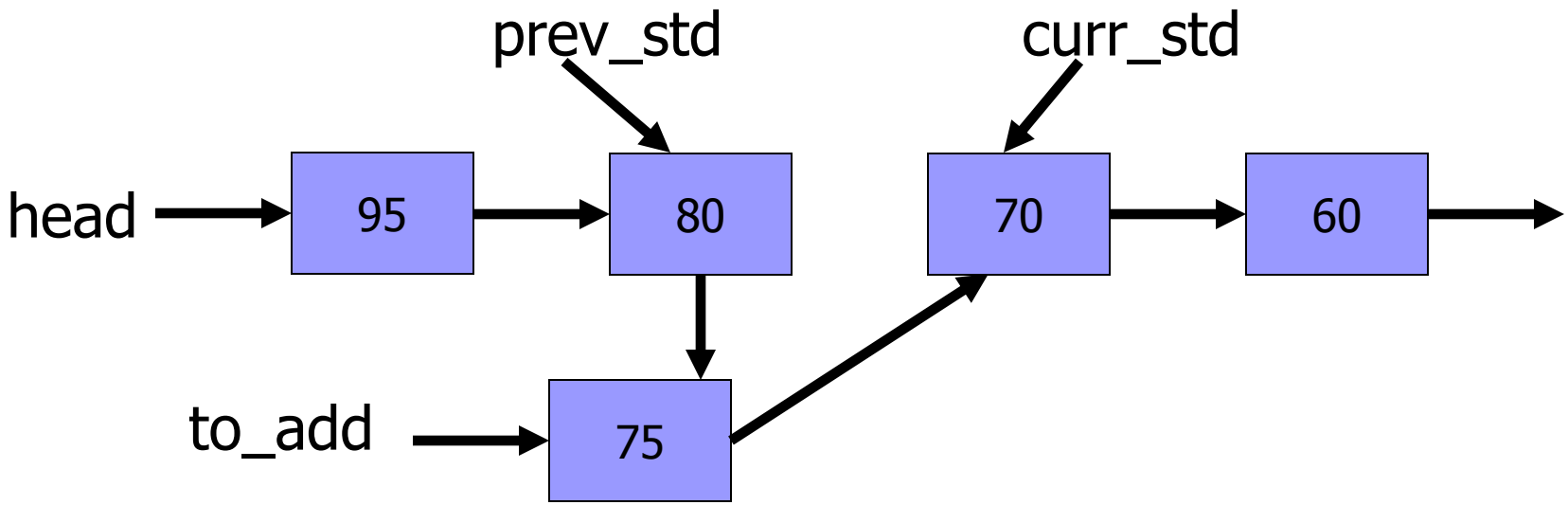
Adding a student – mid / end

```
curr_std = head;
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 head;
```



Adding a student – mid / end

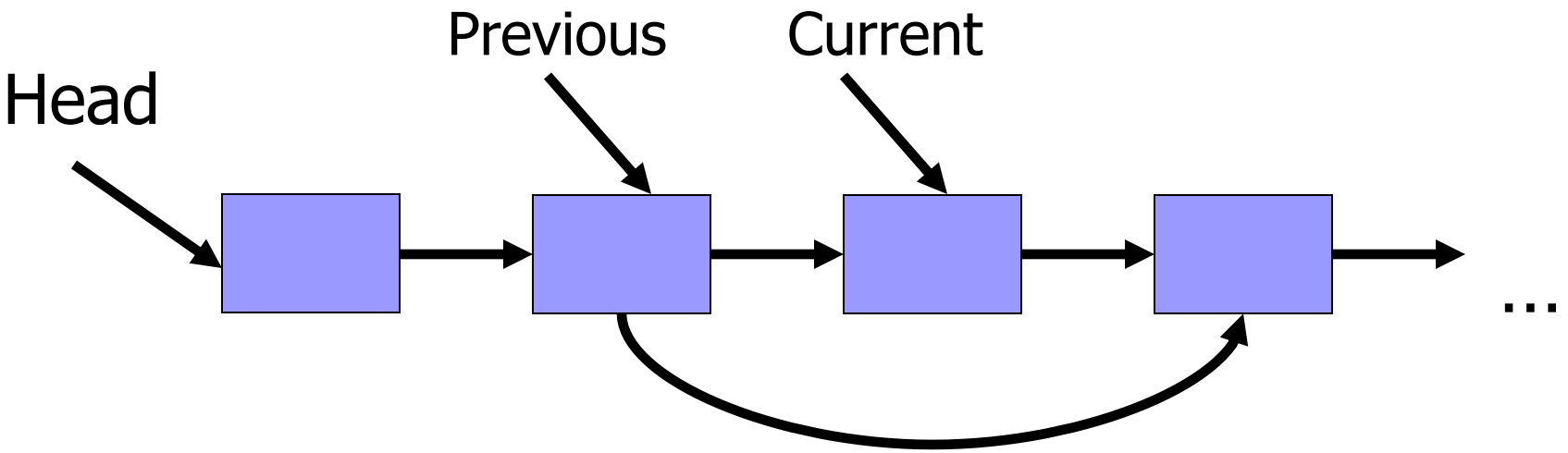
```
curr_std = head;
while (curr_std != NULL  &&  to_add->grade < curr_std->grade)
{
    prev_std = curr_std;
    curr_std = curr_std->next;
}

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



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

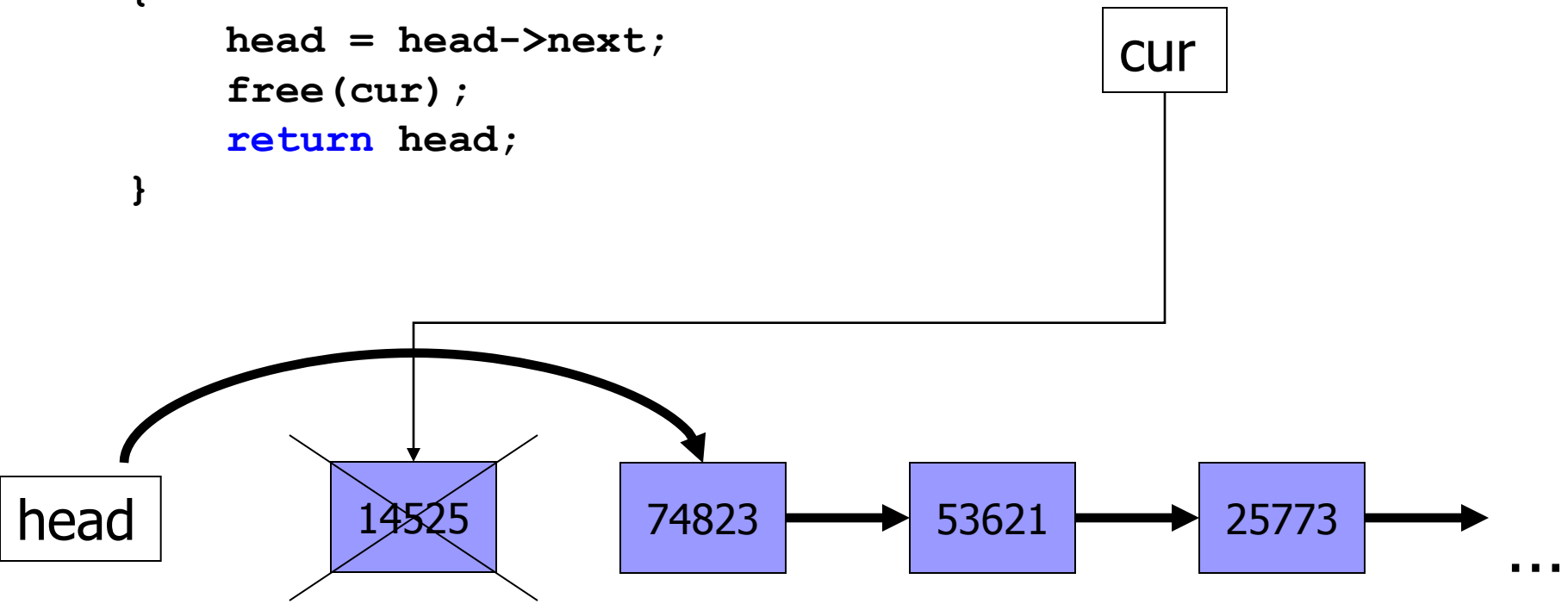


Removing a student – beginning

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

cur = head;

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

ID

14525

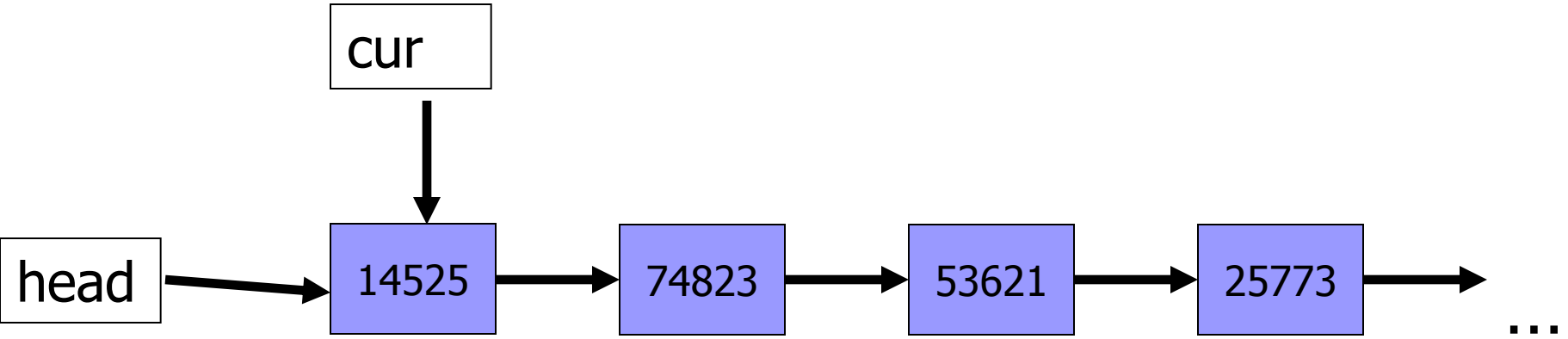


Removing a student – mid list

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

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

return head;
```

ID

53621

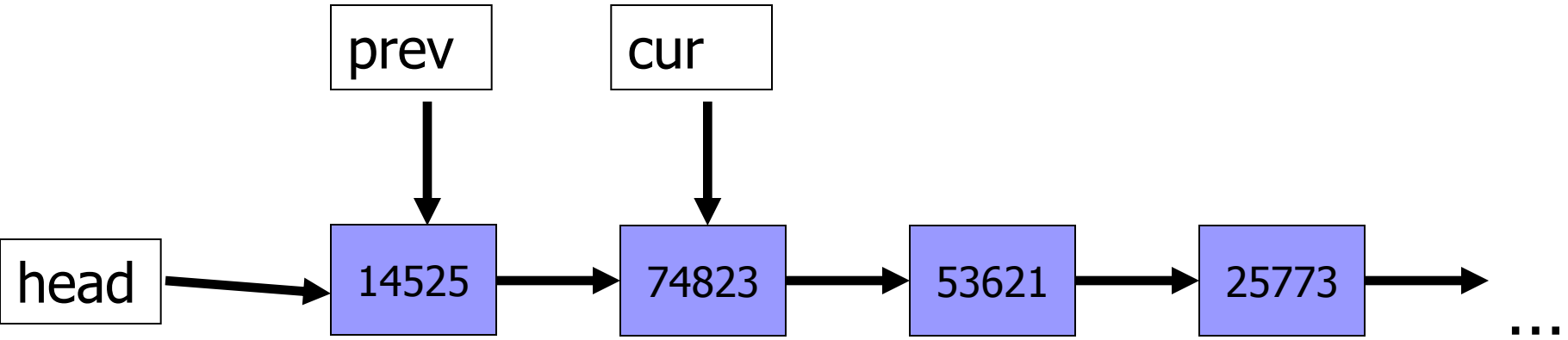


Removing a student – mid list

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

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

return head;
```

ID

53621

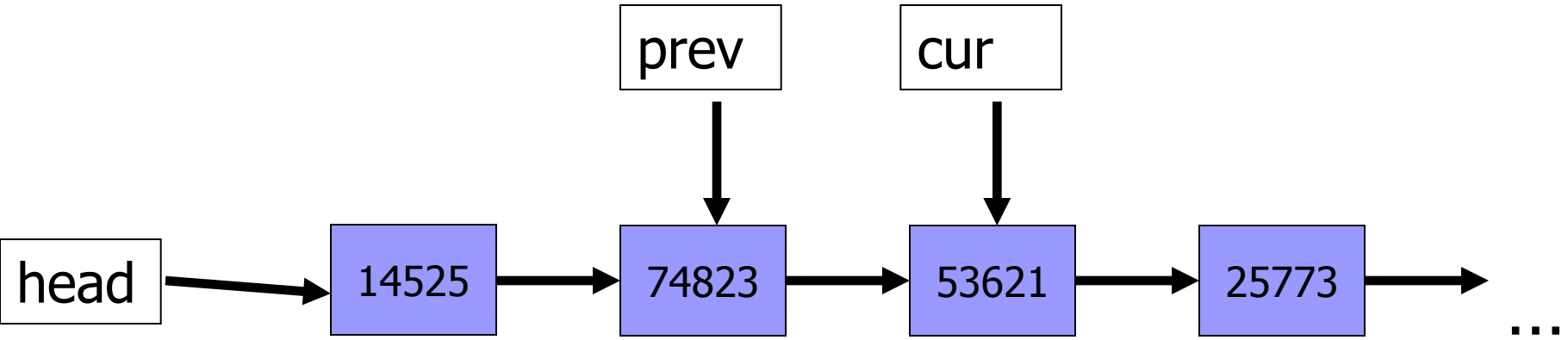


Removing a student – mid list

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

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

return head;
```

ID

53621

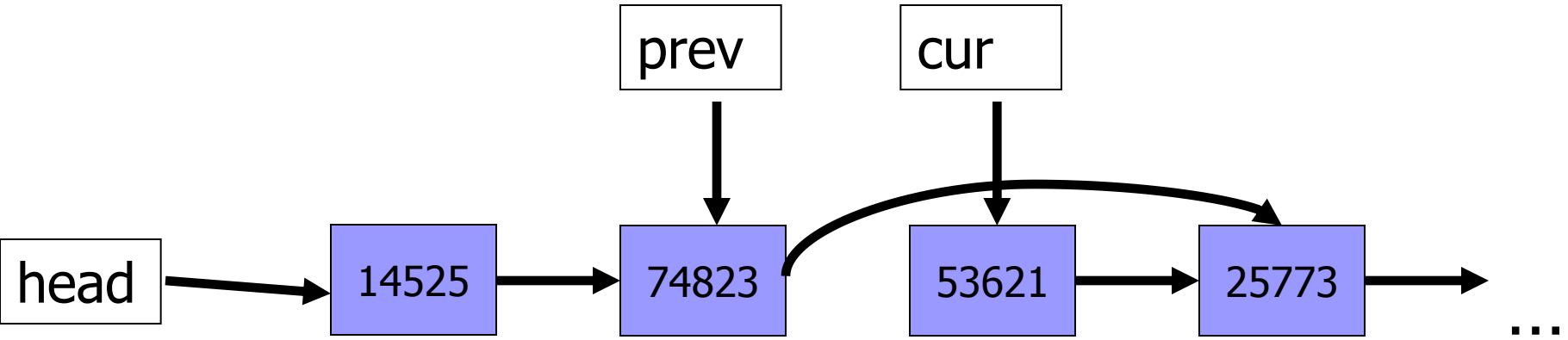


Removing a student – mid list

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

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

return head;
```

ID

53621

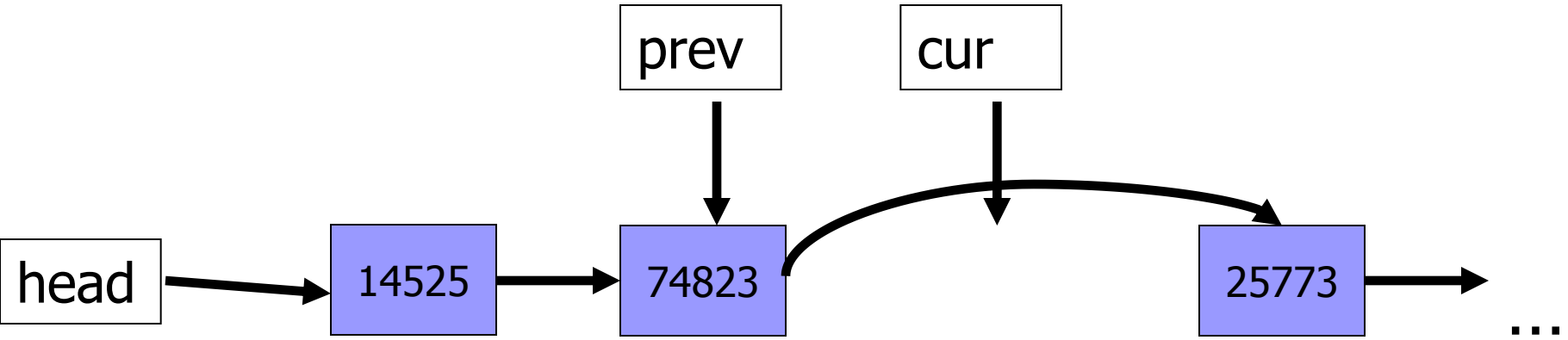


Removing a student – mid list

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

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

return head;
```

ID

53621

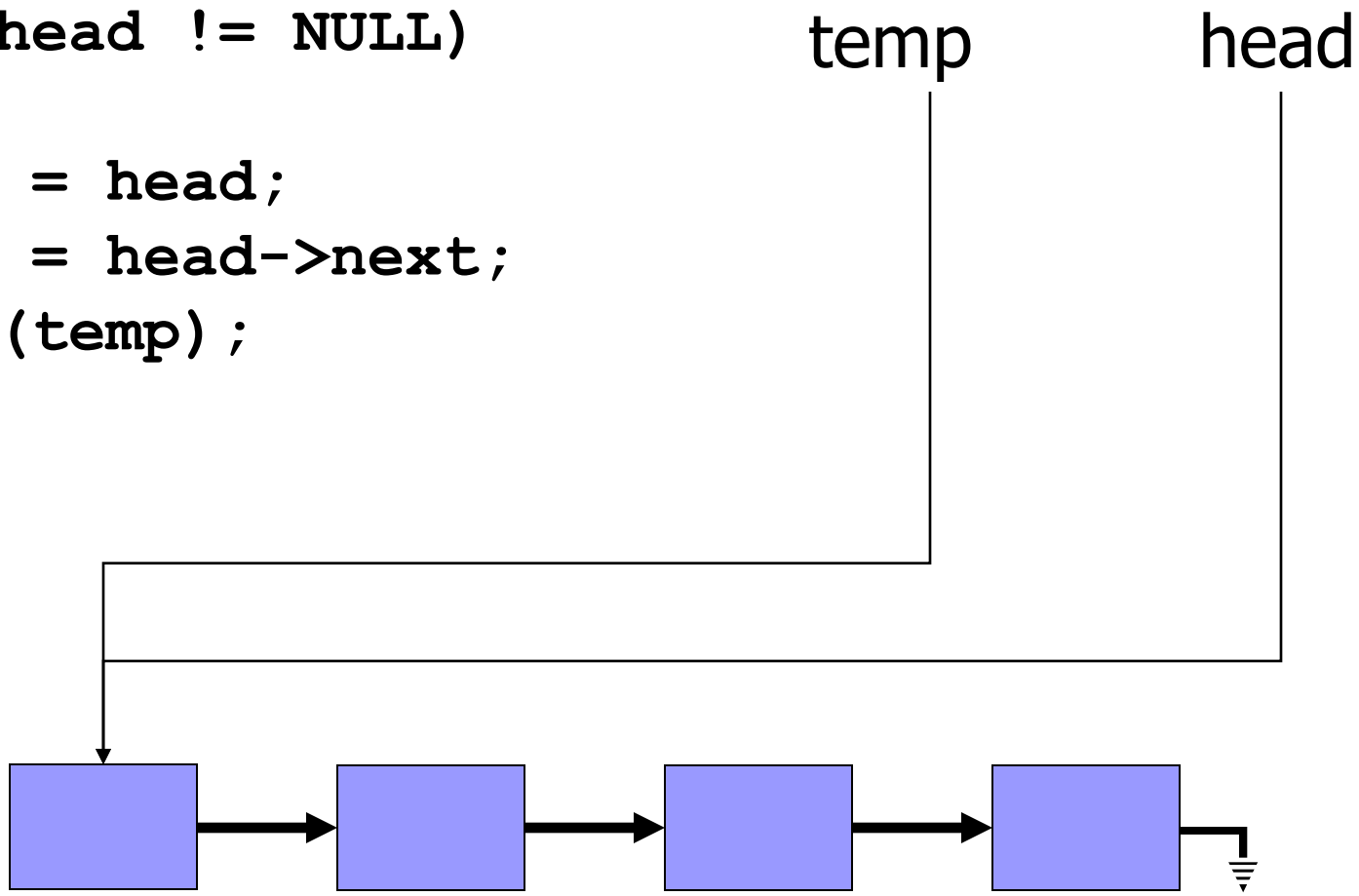


Deallocating all students

```
void free_list(sStudent *head)
{
     sStudent *temp = head;

     while (head != NULL)
     {
          temp = head;
          head = head->next;
          free(temp);
     }
}
```

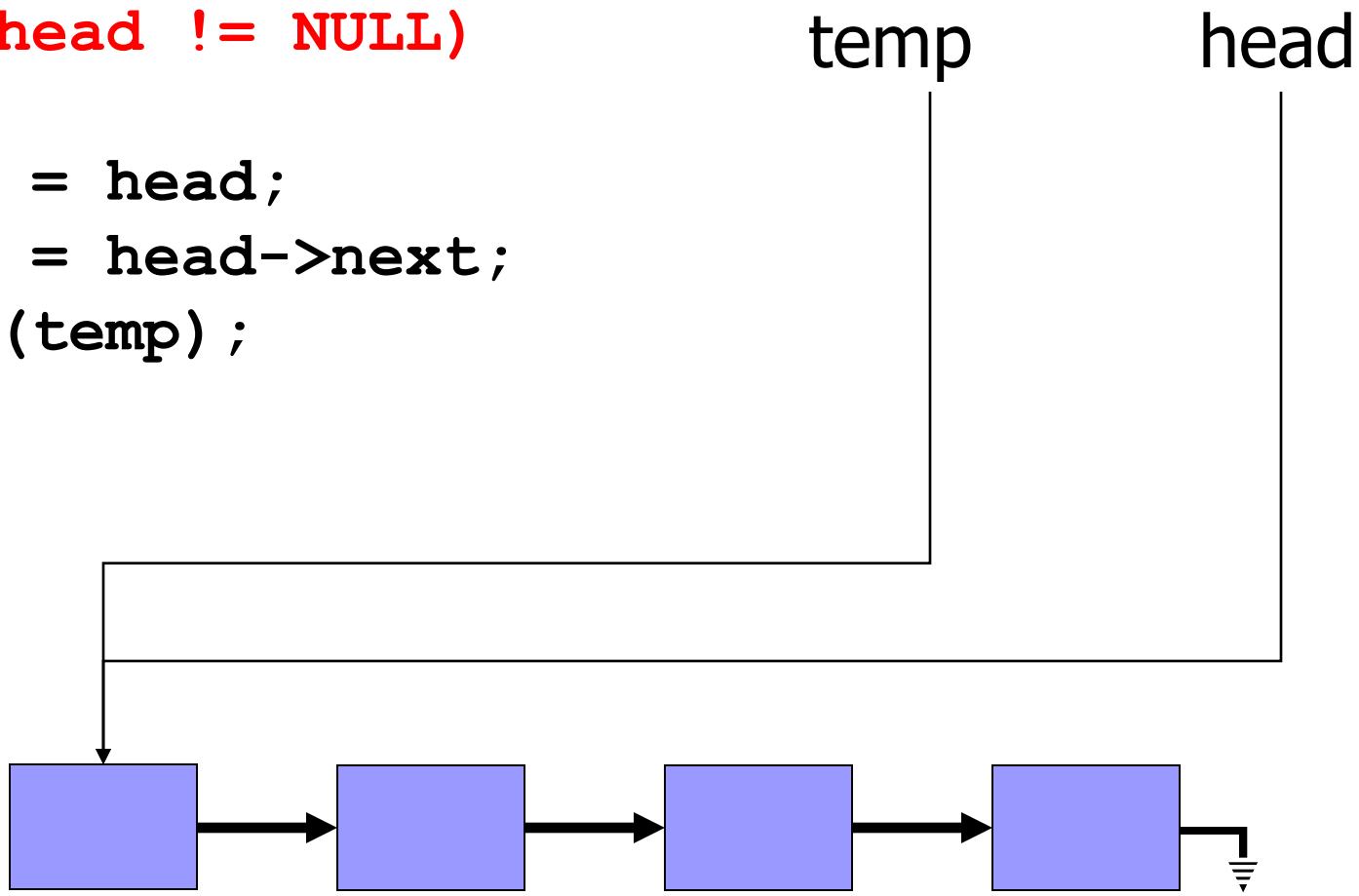
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



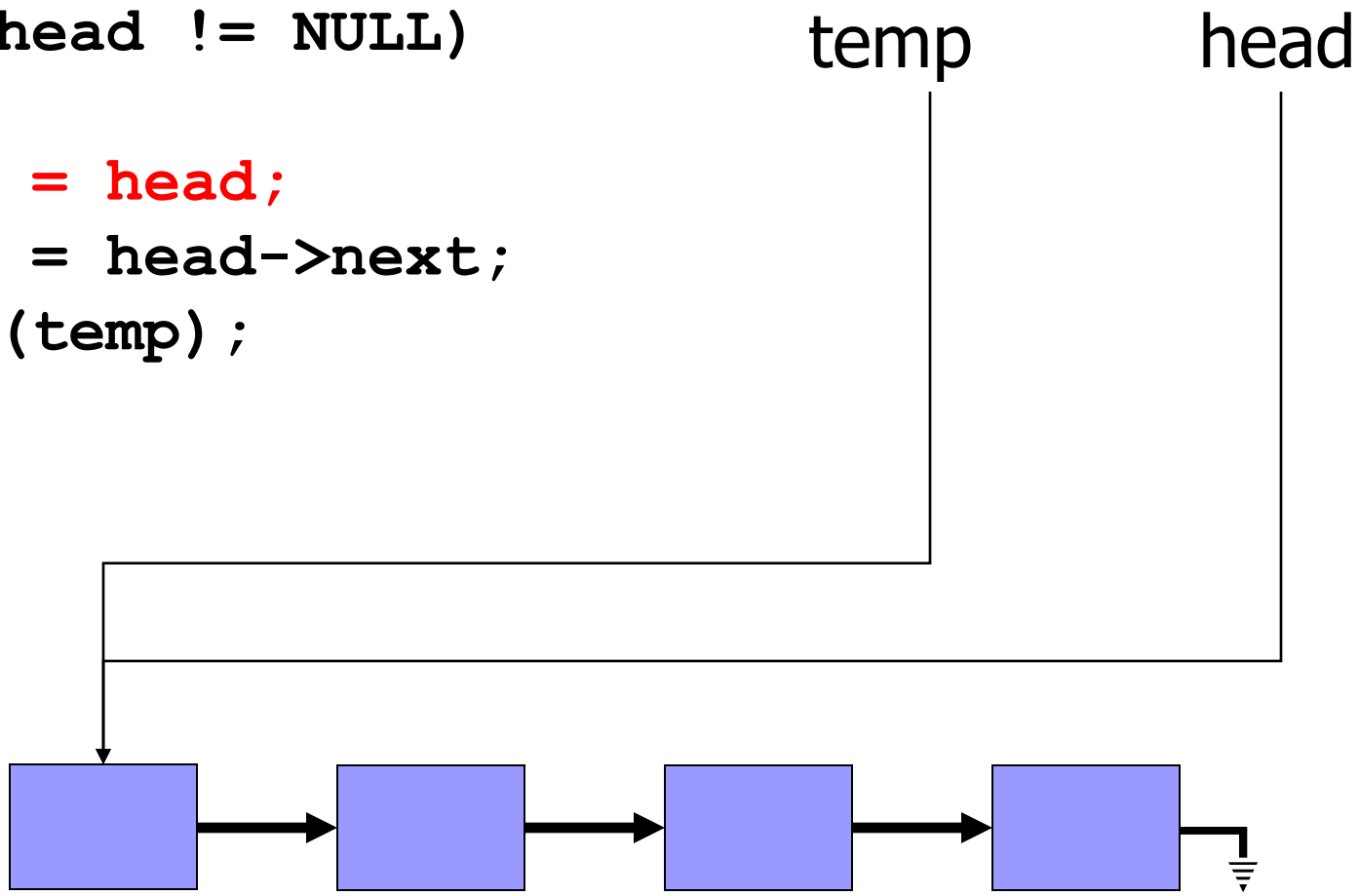
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



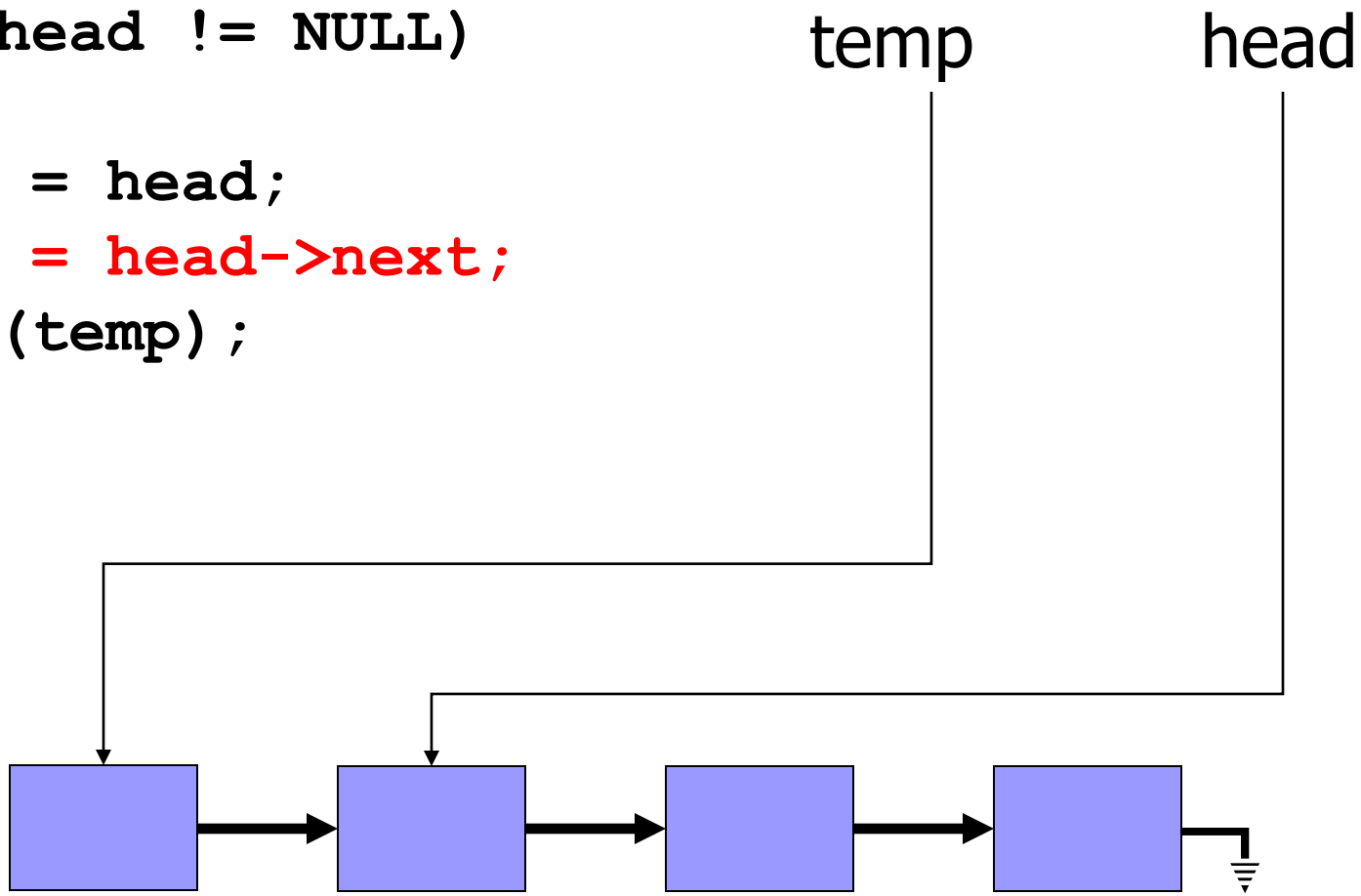
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



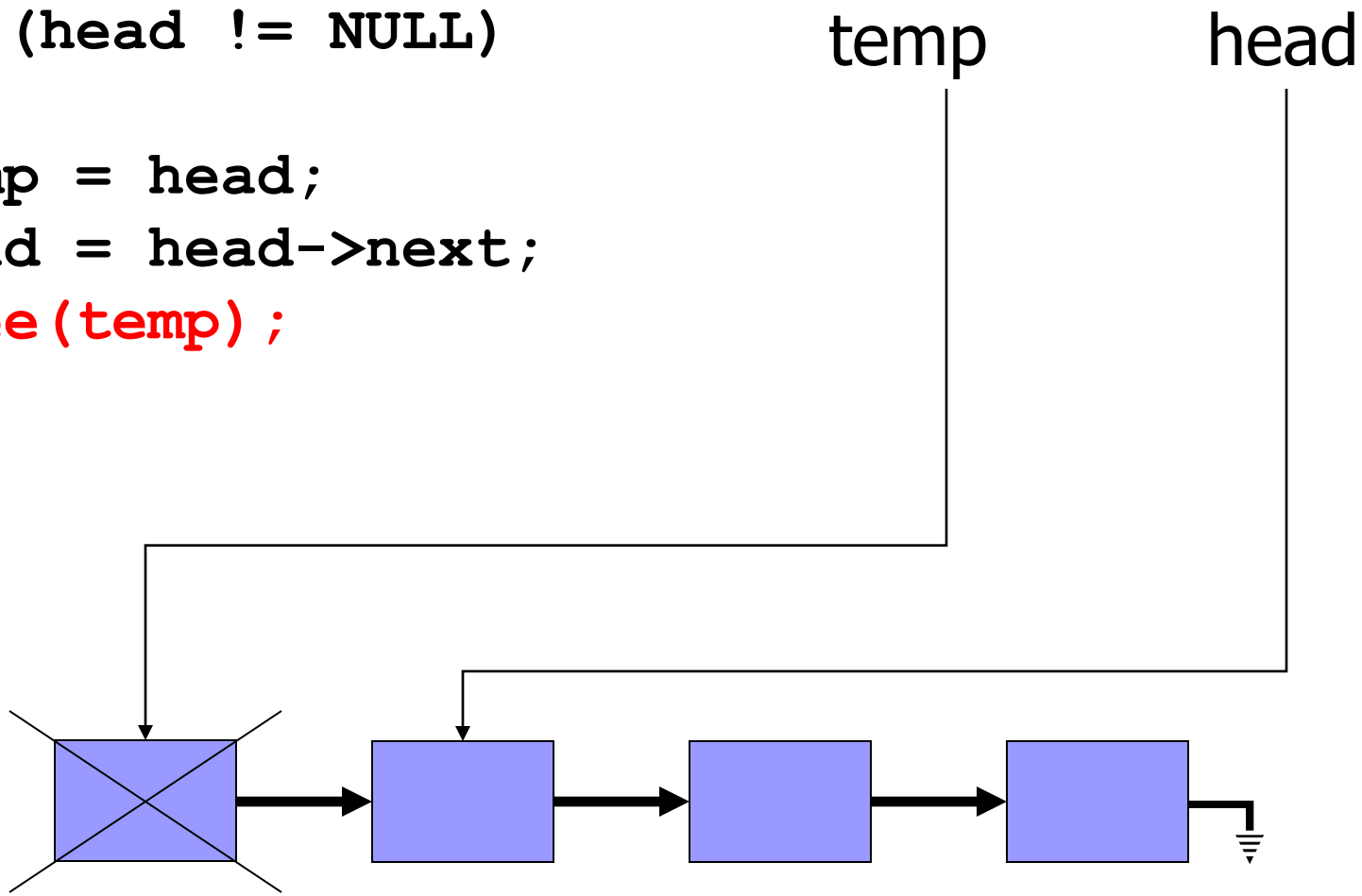
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



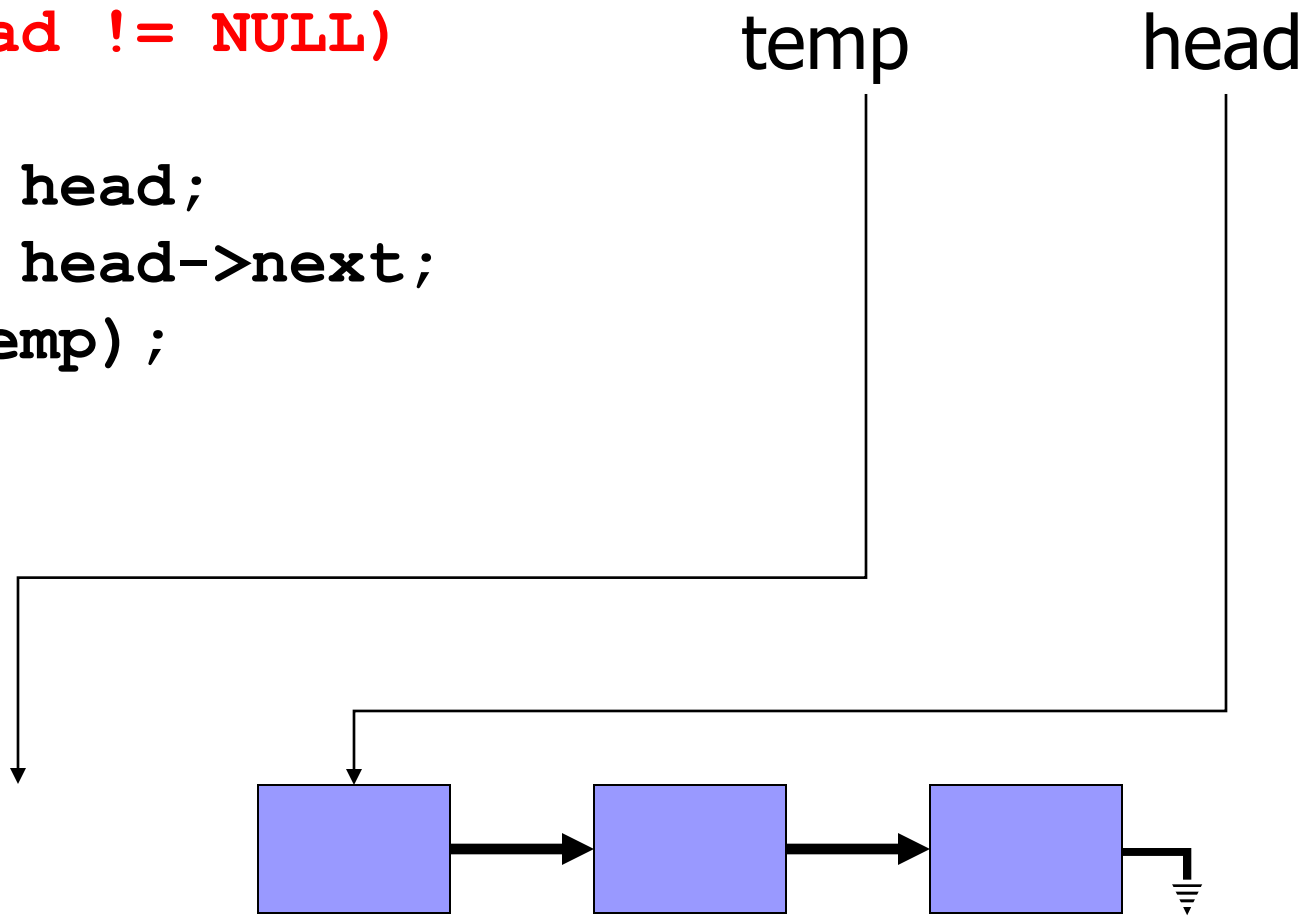
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



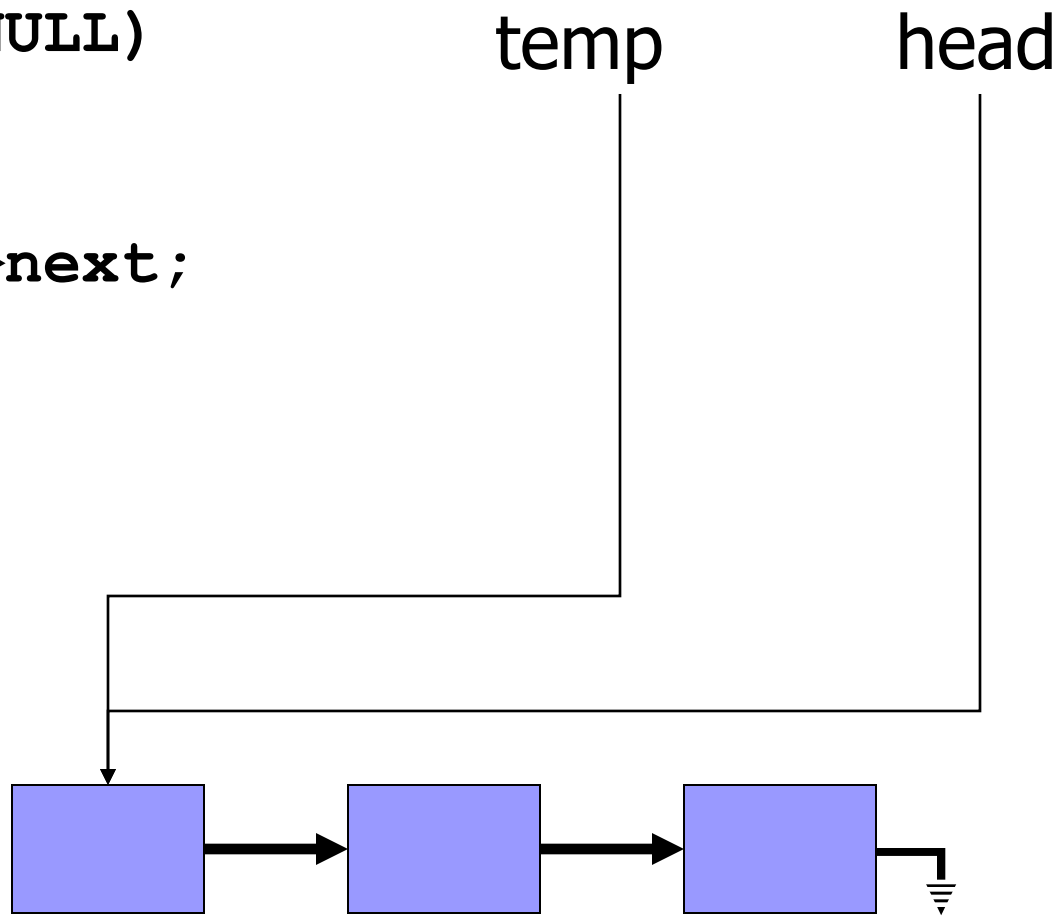
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



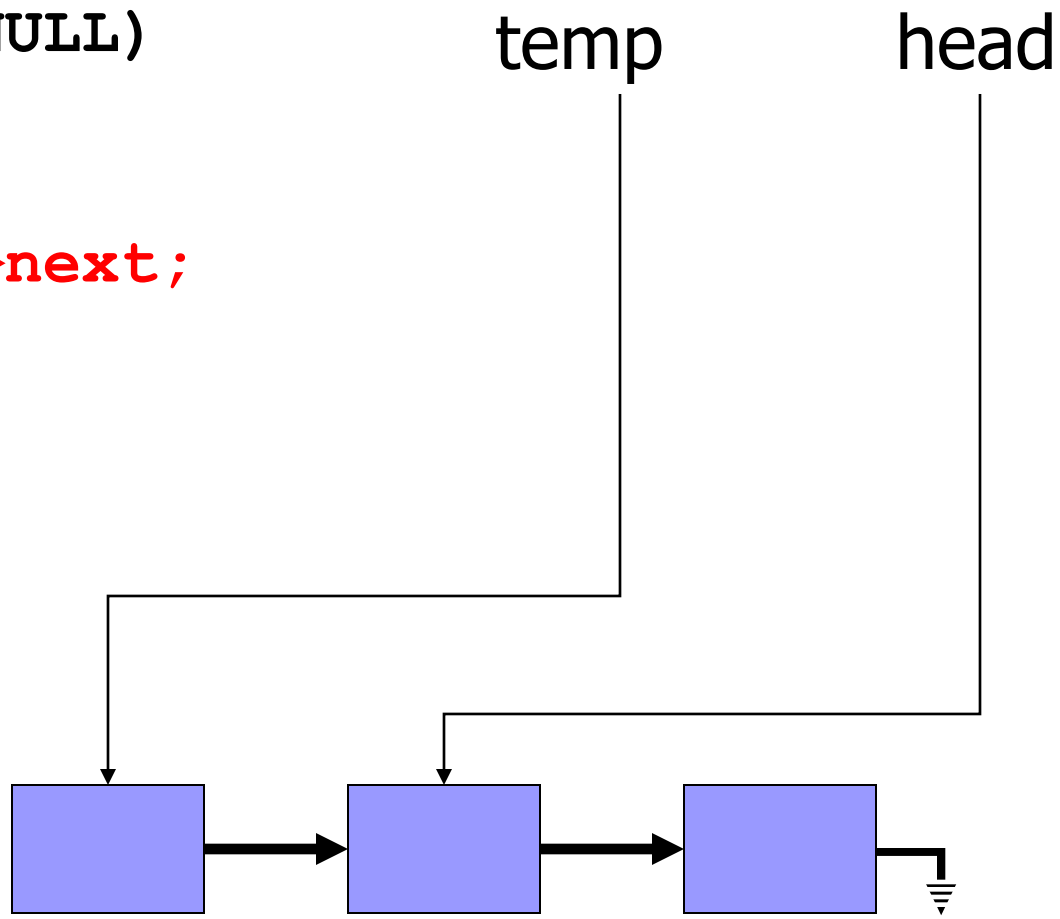
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



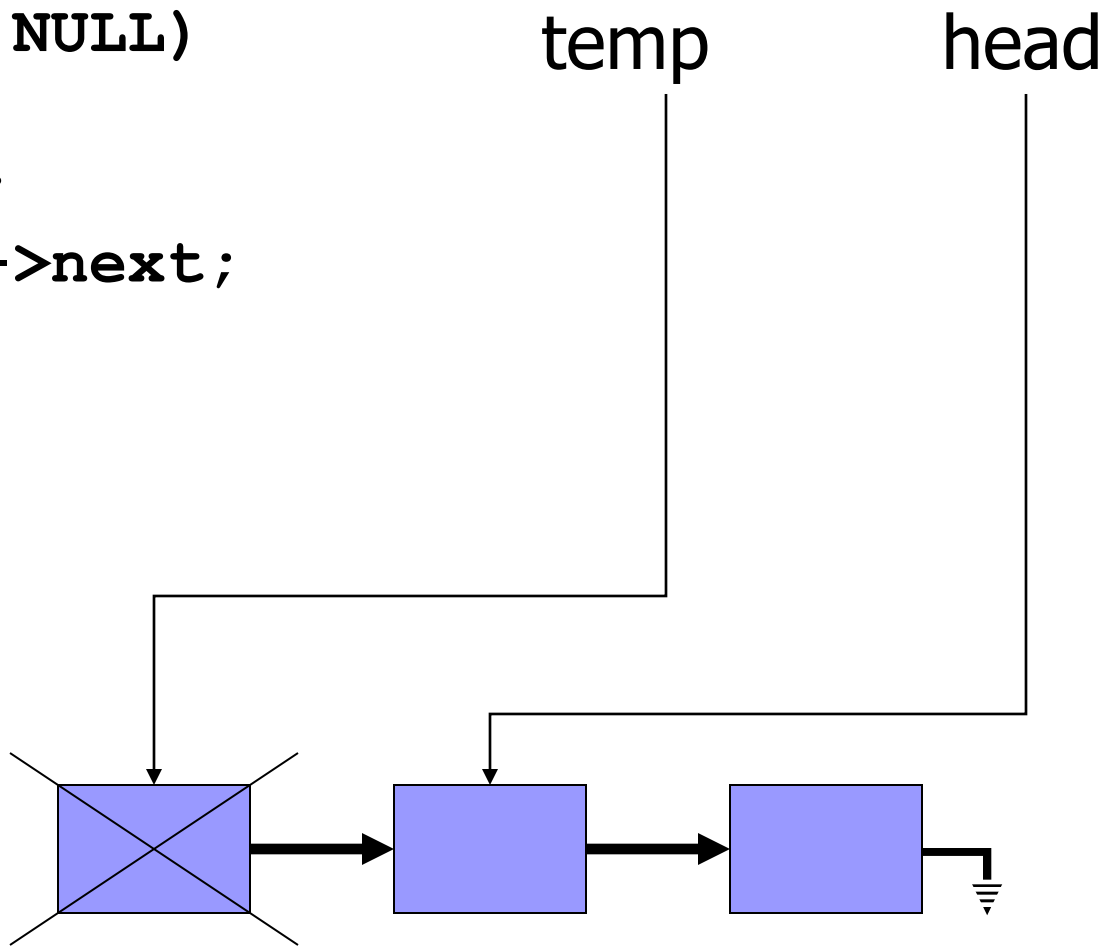
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



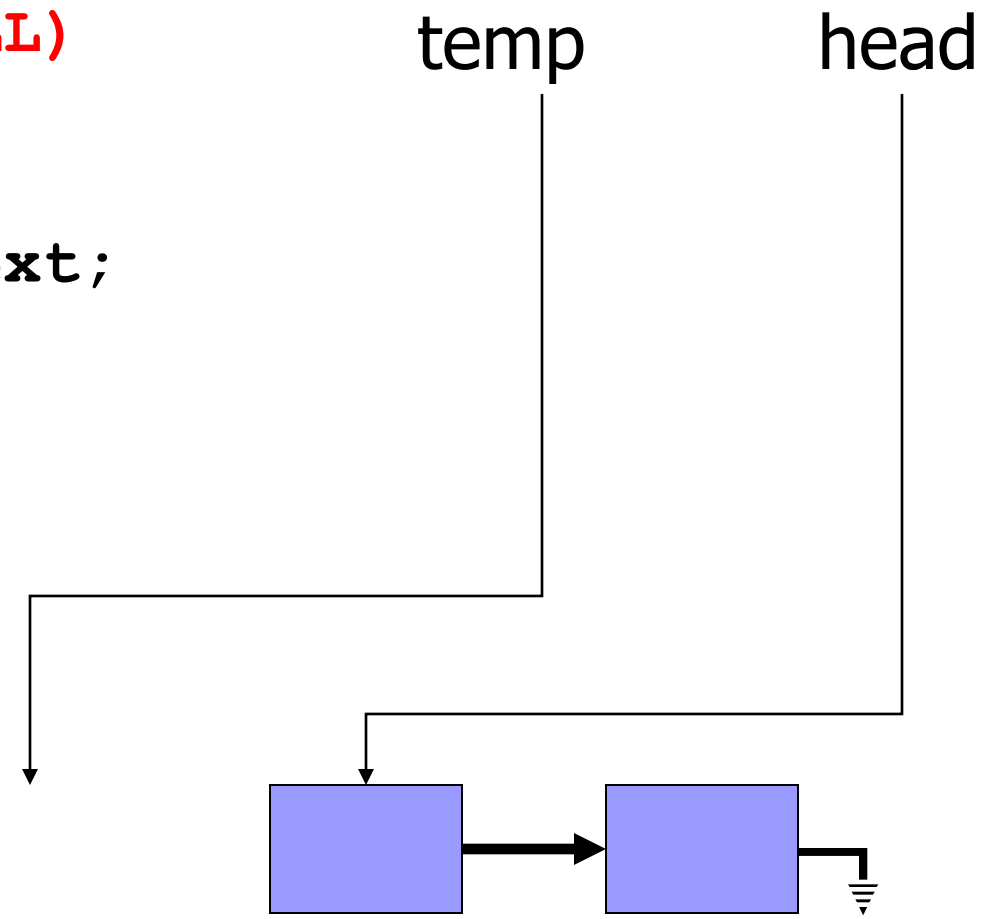
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



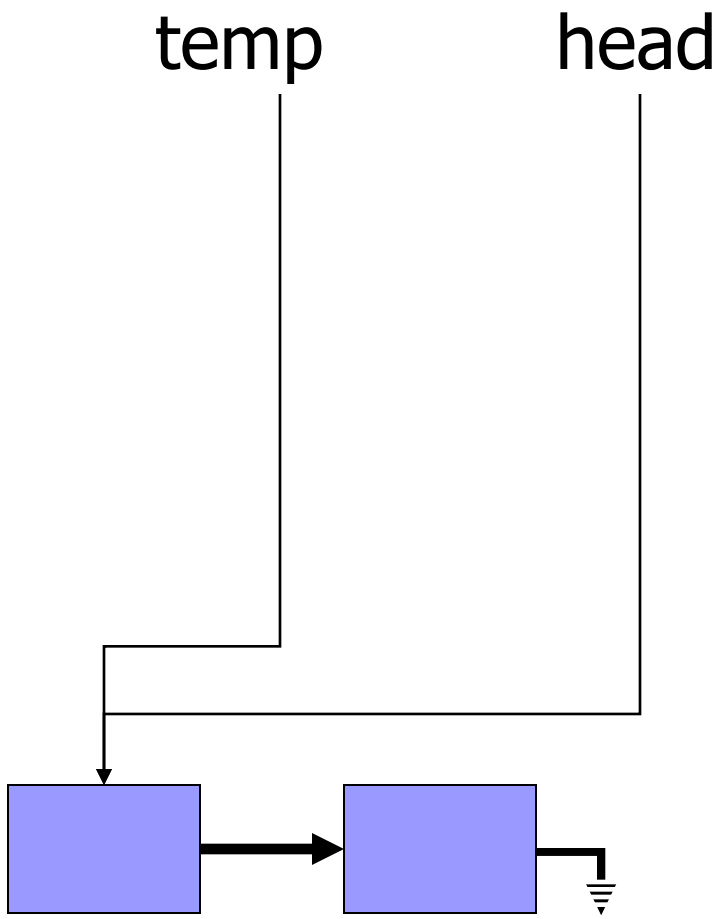
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



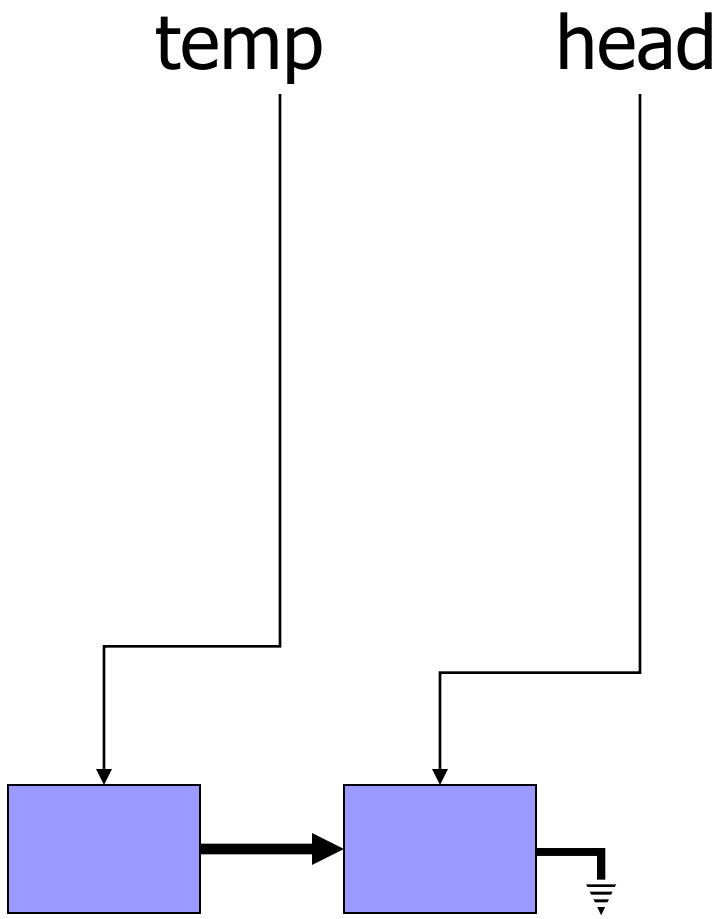
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



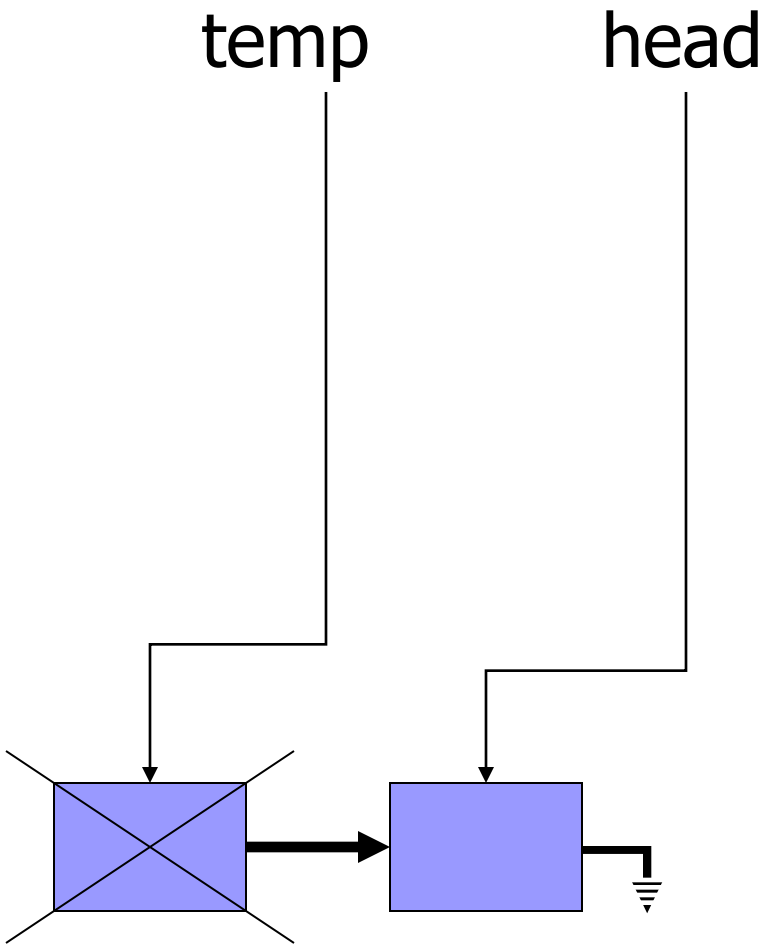
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



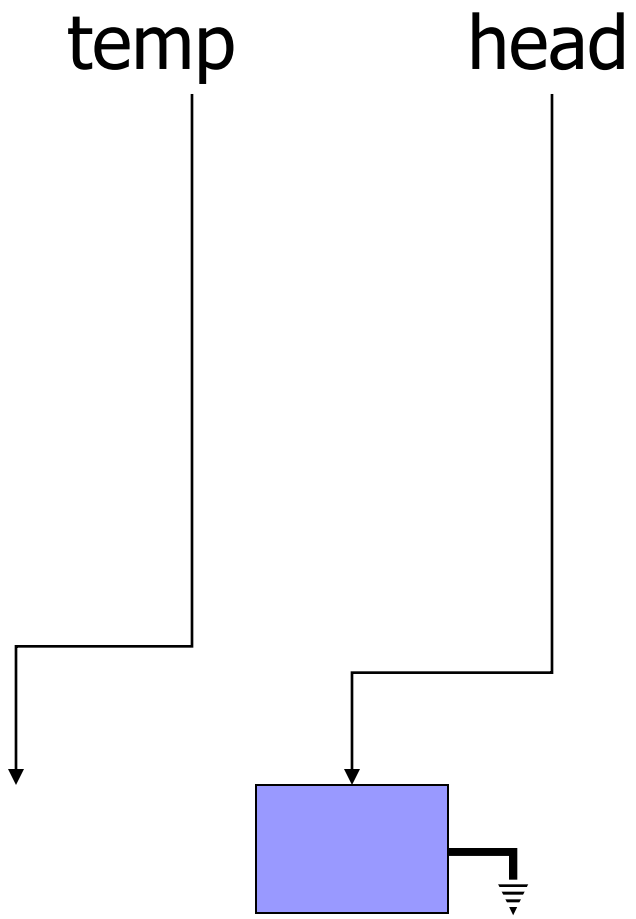
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



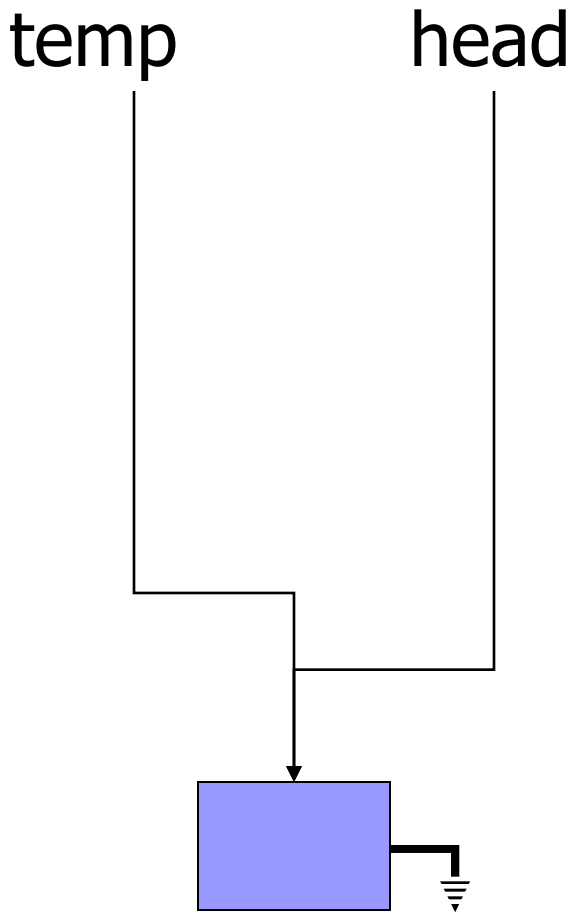
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



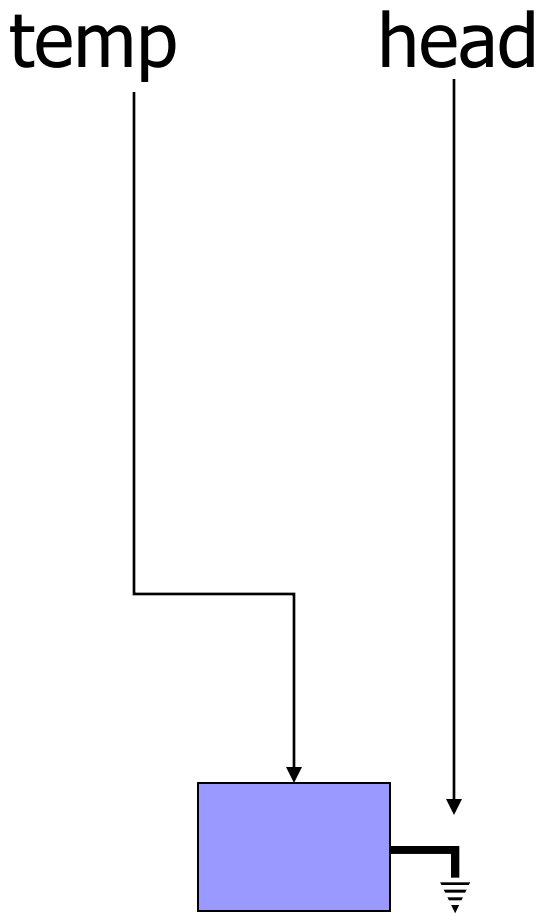
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



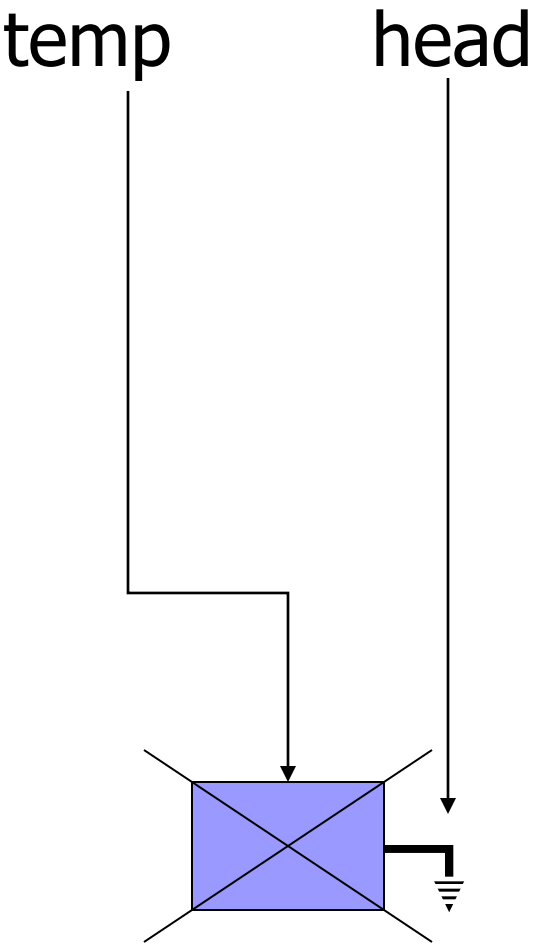
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



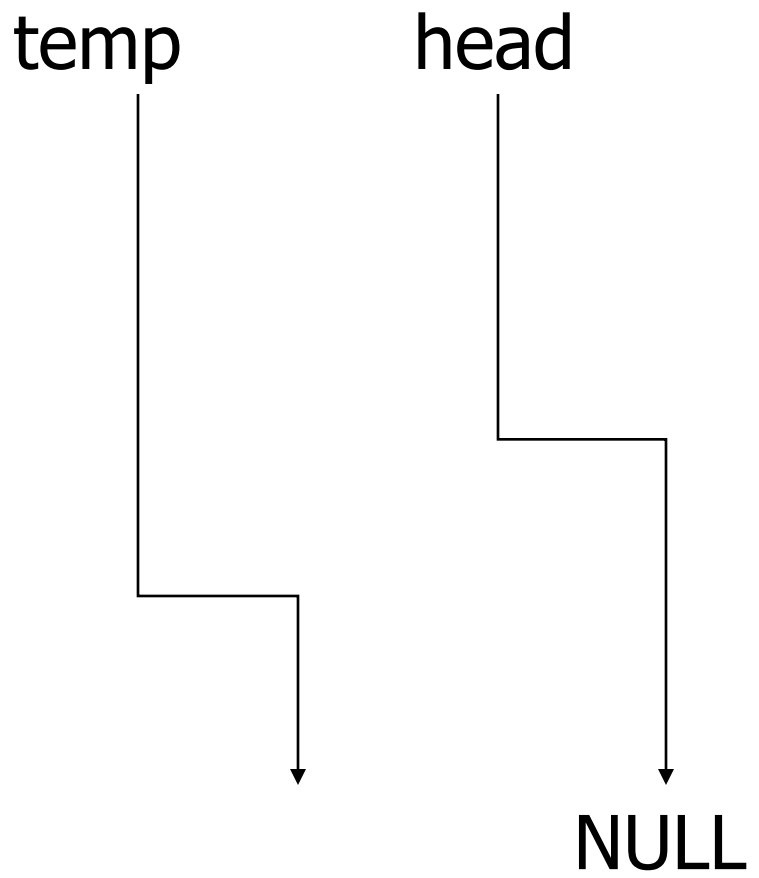
Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```



Deallocating students

```
while (head != NULL)
{
    temp = head;
    head = head->next;
    free(temp);
}
```

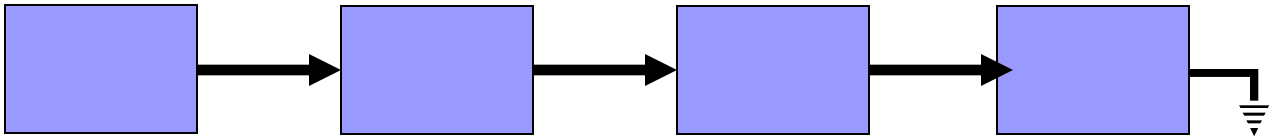


Deallocating students

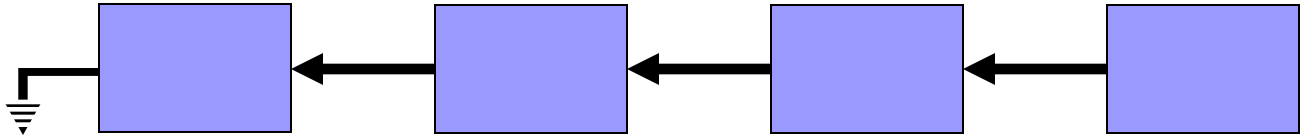
```
void free_list(sStudent *head)
{
    if (head == NULL)
        return;

    free_list(head->next);
    free(head);
}
```

Reverse List

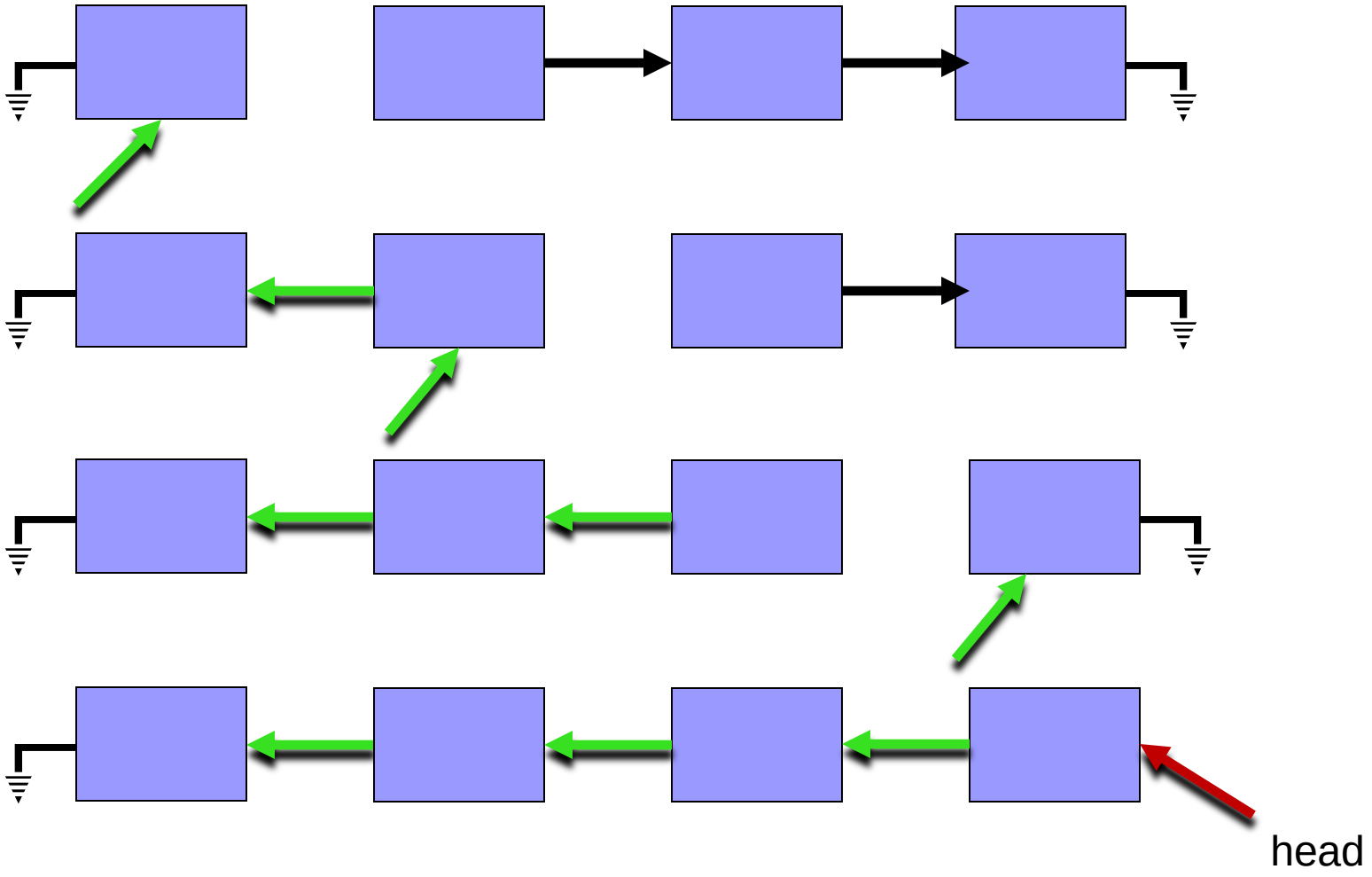
Input:



Output:



Recursive Reverse



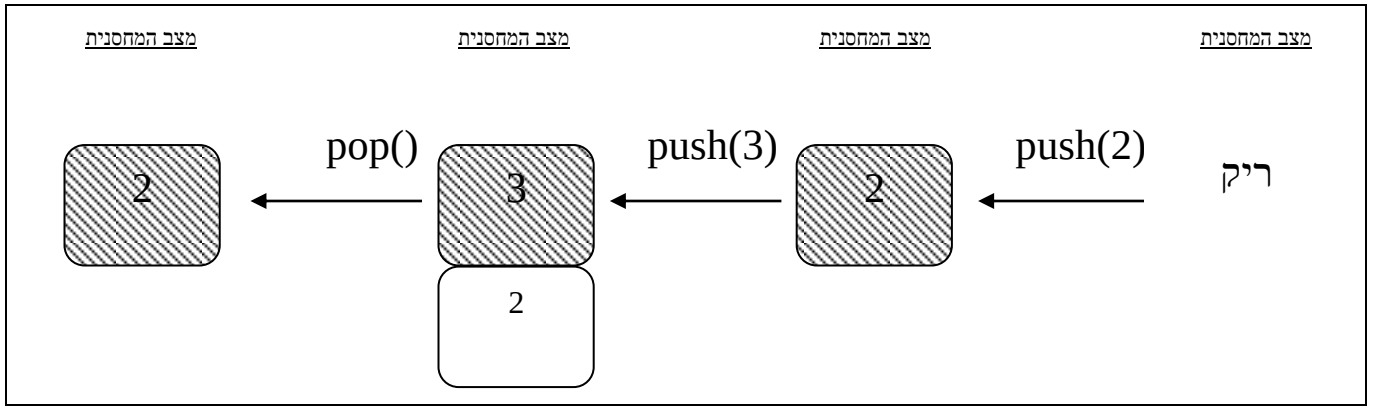
Recursive Reverse - Code

```
Node * reverse_iter(Node * prev, Node* curr)
{
    Node * tmp;

    if (curr == NULL)
        return prev;

    tmp = curr->next;  // keep a link to the rest of the list

    curr->next = prev;  // change direction of current node

    return reverse_iter(curr, tmp);  // reverse the rest of the list
}

Node * reverse(Node * head)
{
    return reverse_iter(NULL, head);
}
```

שאלה 3, סמסטר א' תש"ע, מועד ב'

<u>מחסנית</u> היא מבנה נתונים שבו הערך שנכנס ראשון יוצא אחרון.
נגדיר שתי פעולות על המחסנית:
push - מכניסה ערך לראש המחסנית.
pop - מוציאה את הערך העליון מהמחסנית ומחזירה אותו.

דוגמא: push(2), push(3) ו-pop() על מחסנית ריקה



שאלה 3, סמסטר א' תש"ע, מועד ב'

נממש מחסנית של מספרים שלמים בעזרת רשימה מקושרת.

הגדרת המחסנית:

```
typedef struct stack_t
{
    int size;
    Element *head;
} Stack;
```

שאלה 3, סמסטר א' תש"ע, מועד ב'

יצירת מחסנית חדשה:

```
Stack * createStack()
{
    Stack *s = (Stack*) malloc(sizeof(Stack));
    if (s != NULL)
    {
        s->size = 0;
        s->head = NULL;
    }
    return s;
}
```

שאלה 3, סמסטר א' תש"ע, מועד ב'

יצירת איבר ברשימה:

```
Element * createElement(int data)
{
    Element *e = (Element*) malloc(sizeof(Element));
    if (e != NULL)
    {
        e->data = data;
        e->next = NULL;
    }
    return e;
}
```

מה השדות של המבנה Element?

שאלה 3, סמסטר א' תש"ע, מועד ב'

הכנסת ערך למחסנית (בתחילת הרשימה המקושרת):

```
void push(Stack *s, int data)
{
    Element *e = createElement(data);
    if (e == NULL)
    {
        printf("Fatal error: unable to allocate memory!\n");
        exit(1);
    }
    e->next = s->head;
    s->head = e;
    s->size++;
}
```

שאלה 3, סמסטר א' תש"ע, מועד ב'

בדיקה האם המחסנית ריקה:

```
int isEmpty(Stack *s)
{
    return s->size == 0;
}
```

<u>סעיף א':</u>

הגדירו את המבנה Element. ודאו כי הגדרת המבנה תואמת את השימוש שנעשה בו בפונקציות הנתונות.

שאלה 3, סמסטר א' תש"ע, מועד ב'

```
typedef struct element
{
    int data;
    struct element *next;
} Element;
```

שאלה 3, סמסטר א' תש"ע, מועד ב'

<u>סעיף ב'</u>:

ממשו את הפונקציה pop. שימו לב שעליכם לעדכן את כל השדות הרלוונטיים במבנה המחסנית וכן לדאוג לשחרור זיכרון אם נדרש.

הנחת מימוש: המחסנית אינה ריקה

שאלה 3, סמסטר א' תש"ע, מועד ב'

```
int pop(Stack *s) // assume s is not empty
{
    Element *first = s->head;
    int data = first->data;

    s->head = s->head->next;
    s->size--;

    free(first);
    return data;
}
```

שמירת כתובת וערך האיבר הראשון

עדכון המחסנית

שימוש בכתובת וערך האיבר הראשון

שאלה 3, סמסטר א' תש"ע, מועד ב'

<u>סעיף ג':</u>

ממשו את הפונקציה stackSum.

קלט: מחסנית

פלט: סכום הערכים במחסנית <u>תוך שימוש בפעולות שהוגדרו</u> על המחסנית כדוגמת push, pop ו- isEmpty.

- חל איסור על גישה לרשימת האיברים באופן ישיר.
- לאחר הקריאה לפונקציה המחסנית תהיה ריקה.

שאלה 3, סמסטר א' תש"ע, מועד ב'

```
int stackSum(Stack *s)
{
    int sum = 0;

    while (!isEmpty(s))
        sum += pop(s);

    return sum;
}
```

שאלה 3, סמסטר א' תש"ע, מועד ב'

<u>סעיף ד'</u>:

כתבו תכנית הקולטת ערכים מהמשתמש עד לקבלת ערך שלילי , שומרת אותם במחסנית ולבסוף מחשבת ומדפיסה את סכומם.

- ניתן להניח שקלט המשתמש תקין
- במידה והשתמשתם בהקצאת זיכרון דינמית ודאו שבוצע שיחרור של הזיכרון
- בניקוד סעיף זה יושם דגש על אי שכפול קוד ושימוש בפונקציות שהוגדרו בסעיפים קודמים

שאלה 3, סמסטר א' תש"ע, מועד ב'

```
int main()
{
    int val;
    Stack *s = createStack();
    if (s == NULL) {
        printf("unable to create stack!");
        return 1;
    }
    printf("Please enter input\n");
    scanf("%d", &val);
    while (val >= 0) {
        push(s, val);
        scanf("%d", &val);
    }
    printf("The sum is %d\n", stackSum(s));

    free(s);
    return 0;
}
```

Solution to Class Exercise

```
/* find a student whose id matches the given id */
sStudent* find_student(sStudent *head, char *id)
{
    while (head != NULL) /* go over all the list */
    {
        if (strcmp(head->id, id) == 0) /* same id */
            return head;
        head = head->next;
    }

    /* If we're here, we didn't find it */
    return NULL;
}
```