CONFIDENTIAL

# C Programming Basic – week 3

*For Data Structure and Algorithms*

**Lecturers :**
**Cao Tuan Dung**
**Le Duc Trung**
**Dept of Software Engineering**
**Hanoi University of Technology**

## Today's topics

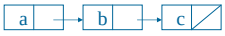
- Self referential structure in C
- Data structure "single linked LIST"
  - Implementation of single linked LIST
  - Algorithm for scanning data
  - Algorithm for inserting, deleting
- Data structure "double linked LIST"
  - Implementation of double linked LIST
  - Algorithm for scanning data
  - Algorithm 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.link=item3.link=NULL;
```






## Implemetation 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



## 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(structnode));
new->element = …
new->next = null;
```

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

## Question 3-1

- 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

7

## Exercise

- 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.

8

## Hint

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

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

9

## Declaration of record structure

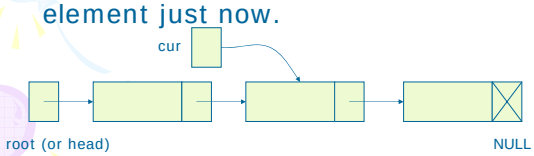
```
struct AddressList {
struct AddressList *next;
struct Address addr;
};
```

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

10

## 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.

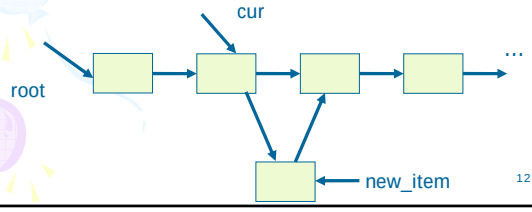



11

## Link list: insertion

- Just after the current position

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




12

## Link list: insertion

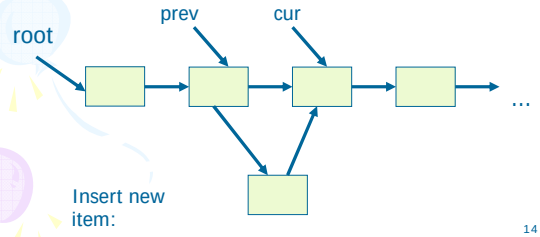
- Just after the current position

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

13

## Linked lists – insertion

- Another case: before the current position

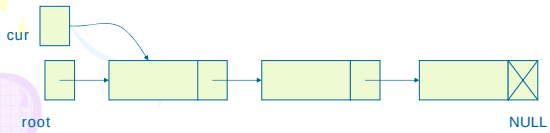



Insert new item:

14

## Traversing a list

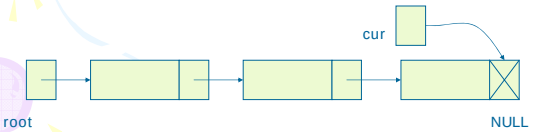
```
for ( cur = root; cur != NULL; cur = cur->next ) {
  showAddress( cur->addr, stdout );
}
```




15

## Traversing a list

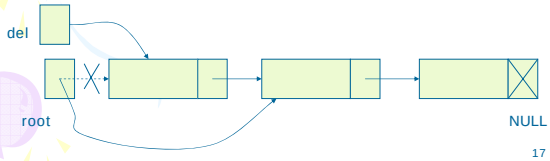
- Changing the value of pointer variable cur in sequence.
- These variables are called “iterator.”
- The traversing is finished if the value is NULL




16

## 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.”

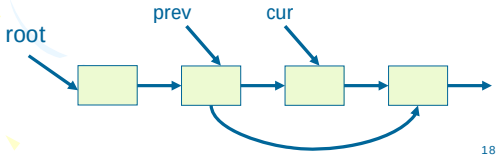



17

## 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);
```




18

## Exercise

- 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.

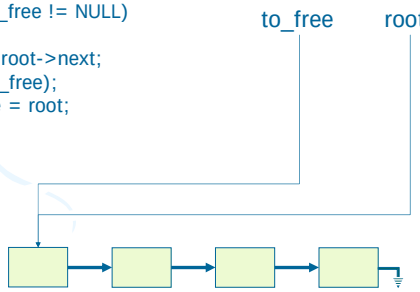
```
struct AddressList *insert   (struct AddressList
  *root, struct Address ad, int n);

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

19

## Freeing a list

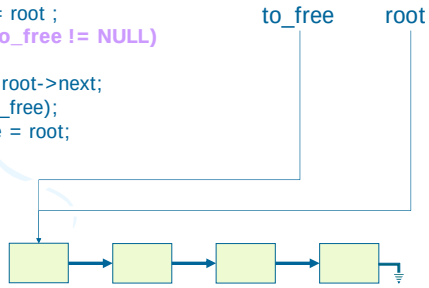
```
to_free = root ;
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




20

## Freeing all nodes of a list

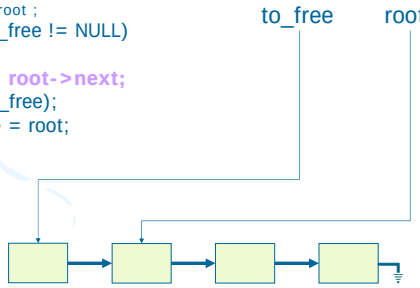
```
to_free = root ;
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




21

## Freeing all nodes of a list

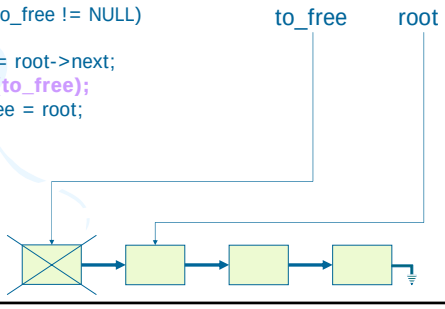
```
to_free = root ;
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




22

## Freeing all nodes of a list

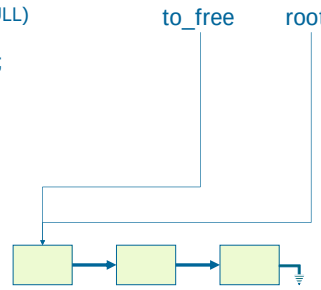
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




23

## Freeing all nodes of a list

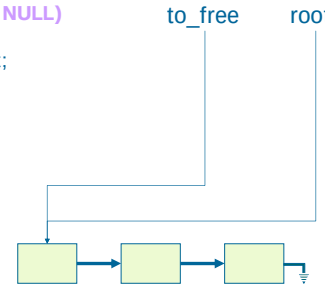
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




24

## Freeing all nodes of a list

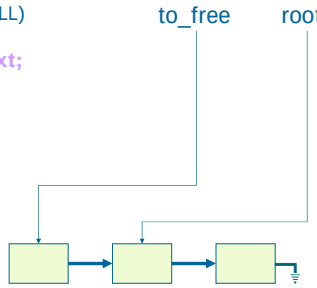
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




25

## Freeing all nodes of a list

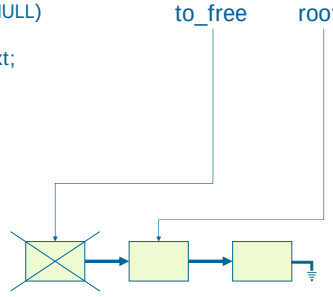
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




26

## Freeing all nodes of a list

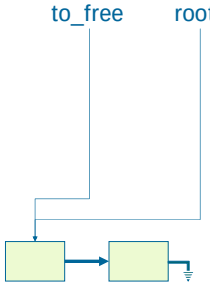
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




27

## Freeing all nodes of a list

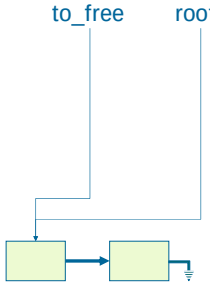
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




28

## Freeing all nodes of a list

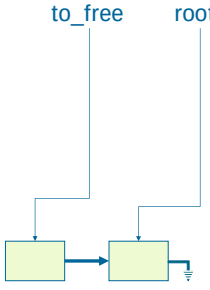
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




29

## Freeing all nodes of a list

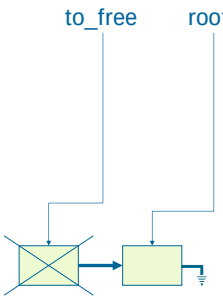
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




30

## Freeing all nodes of a list

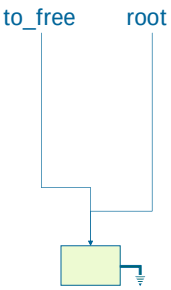
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




31

## Freeing all nodes of a list

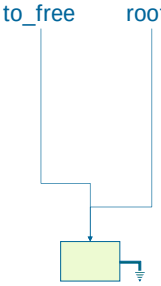
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




32

## Freeing all nodes of a list

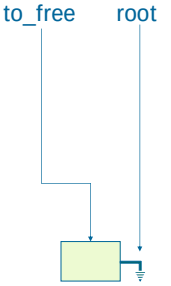
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




33

## Freeing all nodes of a list

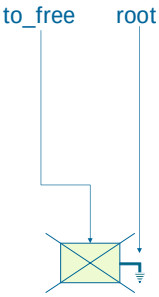
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




34

## Freeing all nodes of a list

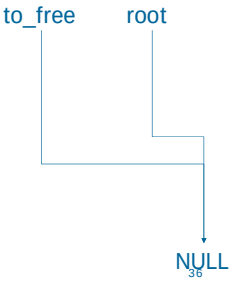
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




35

## Freeing all nodes of a list

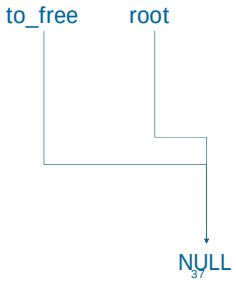
```
while (to_free != NULL)
{
  root = root->next;
  free(to_free);
  to_free = root;
}
```




36

## Freeing all nodes of a list

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




37

## Reverse a list

- 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.

38

## Exercise 3-3

- 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;
```

39

## Exercise 3-3

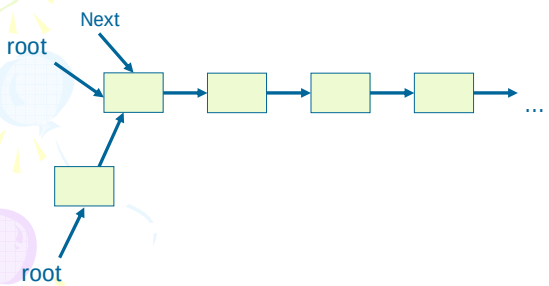
**so that:**

- **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**

- ;

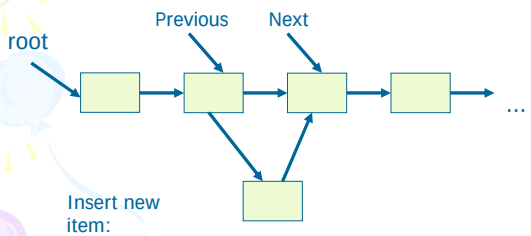
40

## Adding a student - begining




41

## Adding a student – mid/end




42

```
Student *add_student(Student *root, Student *to_add)
{
    Student *curr_std, *prev_std = NULL;

    if (root == NULL)
        return to_add;

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

    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;
}
```

handle empty list

handle beginning

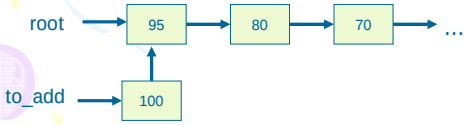
the rest

43

## Adding a student – beginning

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

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





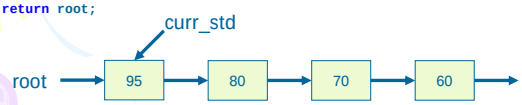


44

## 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;
```



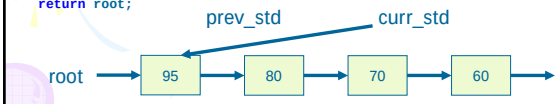


45

## 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;
```





46

## 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;
```



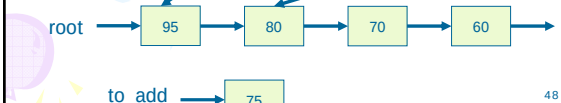


47

## 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;
```





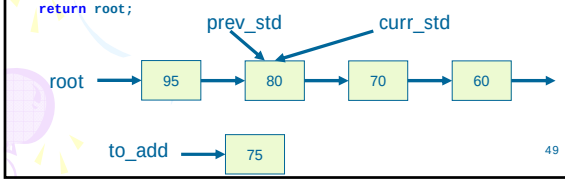


48

## 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;
```

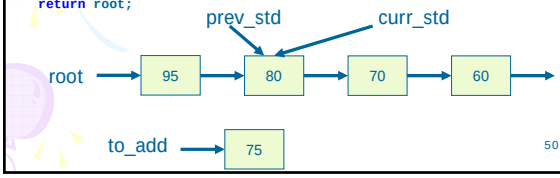



49

## 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;
```

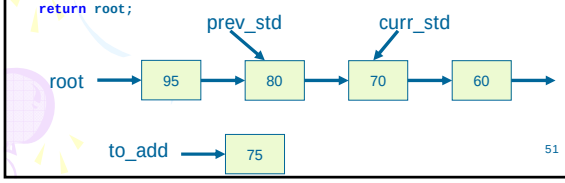



50

## 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;
```

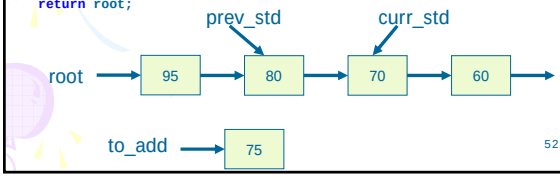



51

## 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;
```

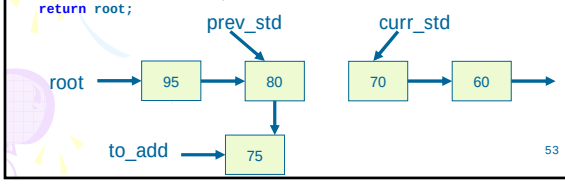



52

## 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;
```

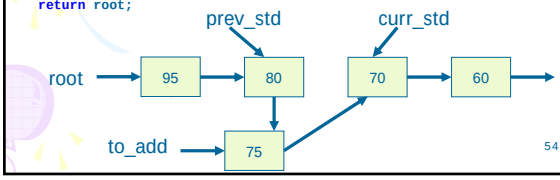



53

## 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;
```




54

## Exercise

- 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

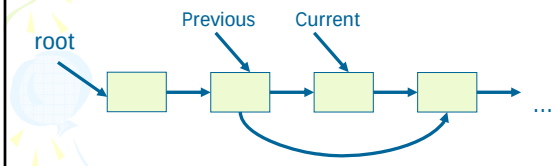
55

## 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`**

56

## Removing a student - reminder




57

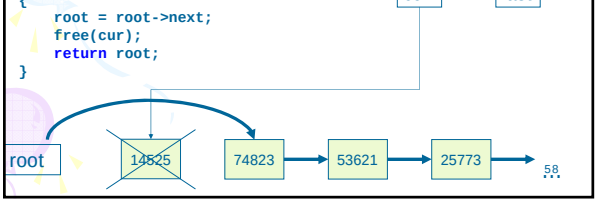
## Removing a student – beginning

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

cur = root;

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

ID: 14525




58

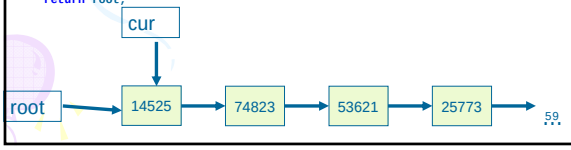
## Removing a student – mid list

```
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




59

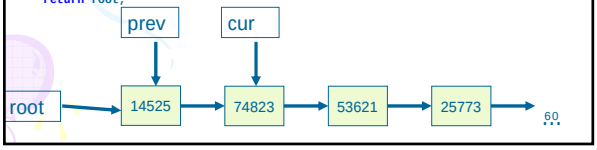
## Removing a student – mid list

```
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




60

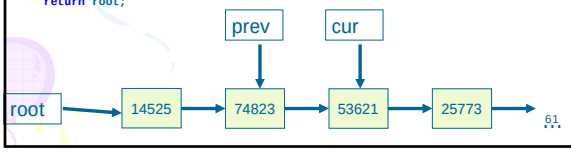
## Removing a student – mid list

```
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




61

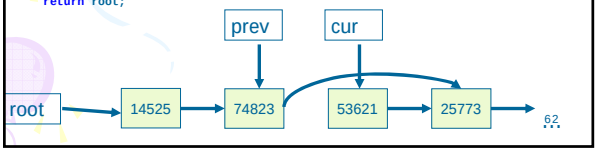
## Removing a student – mid list

```
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




62

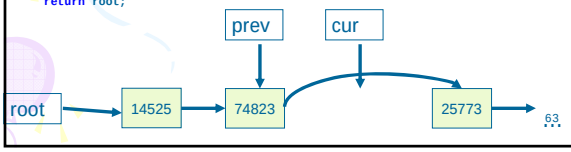
## Removing a student – mid list

```
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




63

## Exercise

- 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

64

## Question

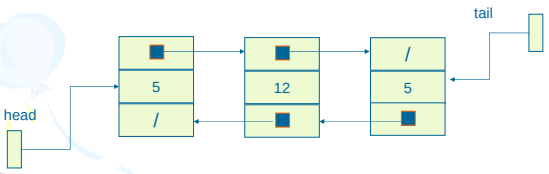
- We are now designing "address list" for mobile phones.
- You must declare a record structure that can keep a name, a phone number, and a e-mail address at least. And you must make the program which can deals with any number of the data.
- Hint: you can organize elements and data structure using following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

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

65

## Double link list

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




66

## Declaration

```
typedef ... ElementType;

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

67

## Initialisation and check for emptiness

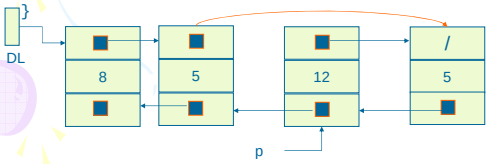
```
void MakeNull_List (DoubleList *DL){
  (*DL)= NULL;
}

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

68

## Delete a node pointed by p

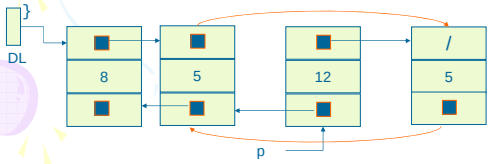
```
void Delete_List (Position p, DoubleList *DL){
 if (*DL == NULL) printf("Empty list");
 else {
   if (p==*DL) (*DL)=(*DL)->Next;
   //Delete first element
   else p->Previous->Next=p->Next;
   if (p->Next!=NULL) p->Next->Previous=p->Previous;
   free(p);
 }
}
```




69

## Delete a node pointed by p

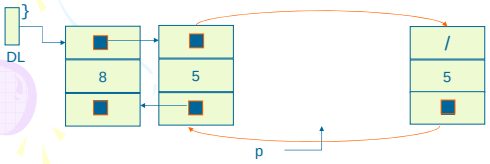
```
void Delete_List (Position p, DoubleList *DL){
 if (*DL == NULL) printf("Empty list");
 else {
   if (p==*DL) (*DL)=(*DL)->Next;
   //Delete first element
   else p->Previous->Next=p->Next;
   if (p->Next!=NULL) p->Next->Previous=p->Previous;
   free(p);
 }
}
```




70

## 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);
 }
}
```




71

## 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;
  }
}
```

72