Generic declaration of a Linked List

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

61

makeNewNode() - generic

```
node* makeNewNode(elementtype addr){
  node* new = (node*)
     malloc(sizeof(node));
  new->element=addr;
  new->next =NULL;
  return new;
}
```

62

insertion just after the current position (generic)

```
void insertAfterCurrent(elementtype e) {

    node* new = makeNewNode(e);
     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;
    }

}
```

63

insertBeforeCurrent

```
void insertBeforeCurrent(elementtype e) {

    node* new = makeNewNode(e);
     if ( root == NULL ) {
    /* if there is no element */
        root = new;
        cur = root;
        prev = NULL;
    } else {

        new->next=cur;
         /* if cur pointed to first element */
        if (cur==root) {
                root = new; /* nut moi them vao tro thanh dau danh
    sach */
        }
        else prev->next = new;
        cur = new;
    }
}
```

64

Read/Write specific data

```
elementtype readData(){
  elementtype res;
  printf("name:");gets(res.name);
  printf("tel:");gets(res.tel);
  printf("email:");gets(res.email);
   return res;
}
void printData(elementtype res){

  printf("%15s\t%10s\t%20s\n",res.n
  ame, res.tel, res.email);
}
```

65

Traversing a List (generic)

```
void traverseList() {
  node* p;
  for (p = root; p !=
  NULL; p = p->next ) {
    printData(p->element);
}
```

66

Generic version

```
node* list_reverse (node* root)
{
  node *cur, *prev;
  cur = prev = NULL;
  while (root != NULL) {
      cur = root;
      root = root->next;
      cur->next = prev;
      prev = cur;
  }
  return prev;
}
```

67

Delete an element with given data

- Write API of linkedlist library
- void deleteElement(elementtype e);
- Delete the first element which has the info field e.

68

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.**

```
Node *insertAtPosition(Node *root, elementtype
  ad, int n);
Node *deleteAtPosition(Node *root, int n);
```

```
Kết quả insert trả về contrỏ trỏ tới nút vừa thêm
  trong danh sách. Kết quả delete trả về root của
  danh sách sau khi xóa
```

69

generic version

```
Node * insertAtPosition(elementType ad, int n){

  // fill in the code here
}
```

70

delete generic version

```
Node * deleteAtPosition(int n){

   // fill in the code here
}
```

71

Summary of the functionalities of the PhoneDB exercises (Singly Linke List)

- **1. Import from PhoneDB.dat (insertafter)**
- **2. Display (traverse)**
- **3. Add new phone (insertbefore/after)**
- **4. Insert at Position**
- **5. Delete at Position**
- **6. Delete current**
- **7. Delete first**
- **8. Search and Update: (Search by model – update all field of information)**
- **9. Divide and Extract (split): Output is the content of two sublists.**
- **10. Reverse List**
- **11. Save to File**
- **12. Quit(Free)**

72

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;
```

73

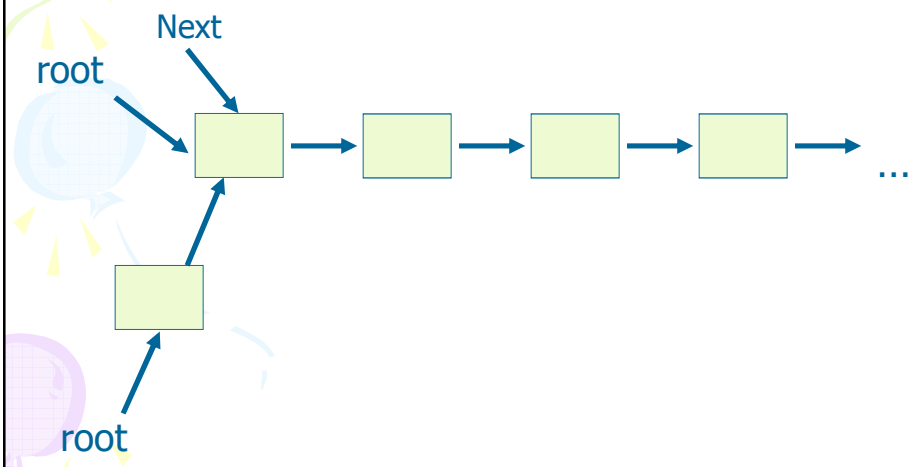
Exercise 3-3

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
- ;

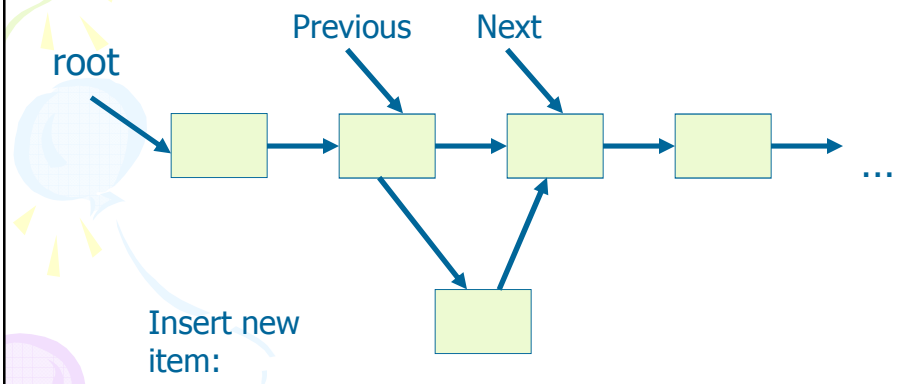
74

Adding a student - begining



75

Adding a student – mid/end



76

```
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

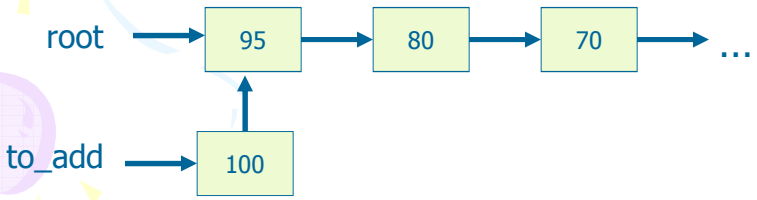
the rest

77

Adding a student – beginning

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

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

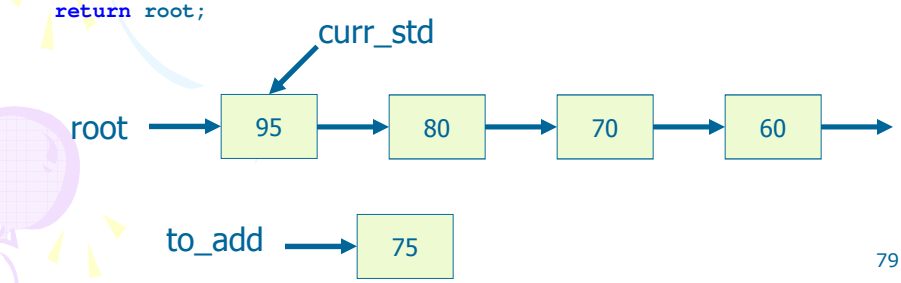


78

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;
```

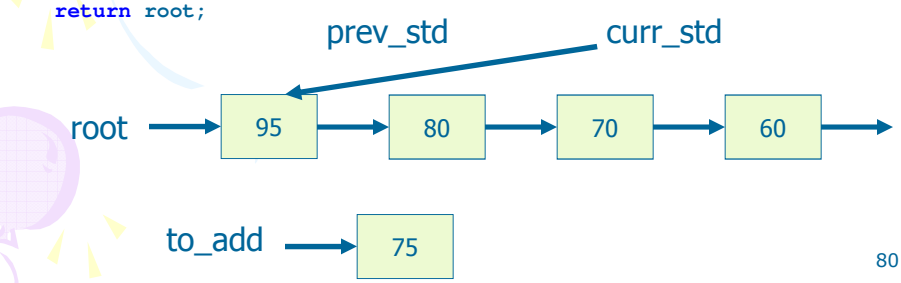


79

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;
```

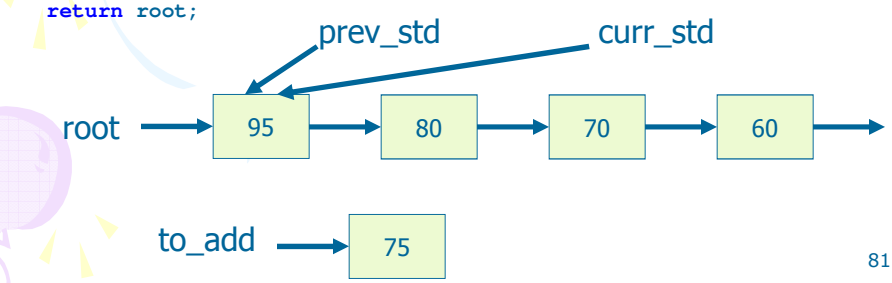


80

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;
```

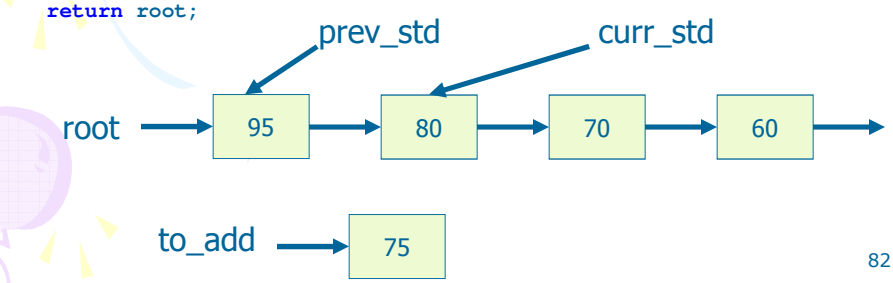


81

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;
```

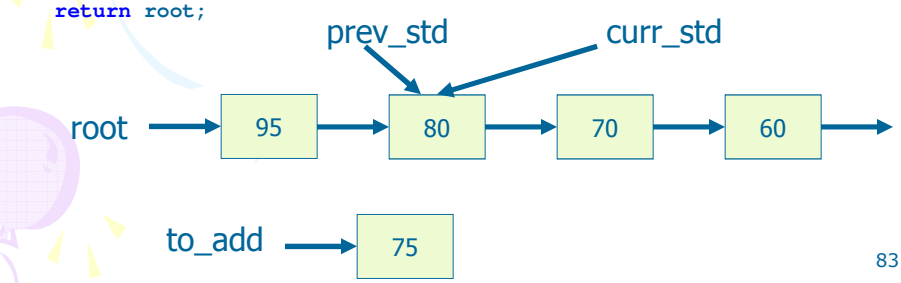


82

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;
```

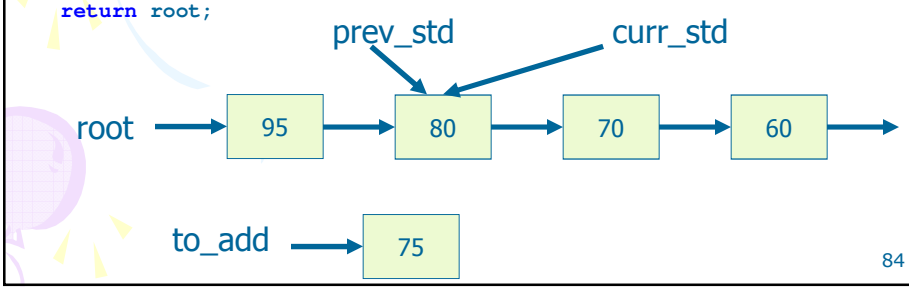


83

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;
```

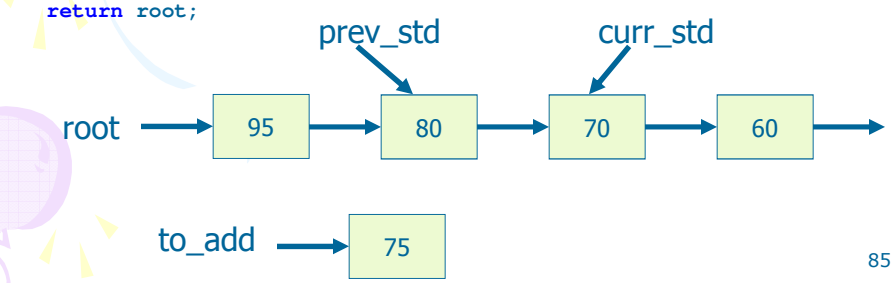


84

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;
```

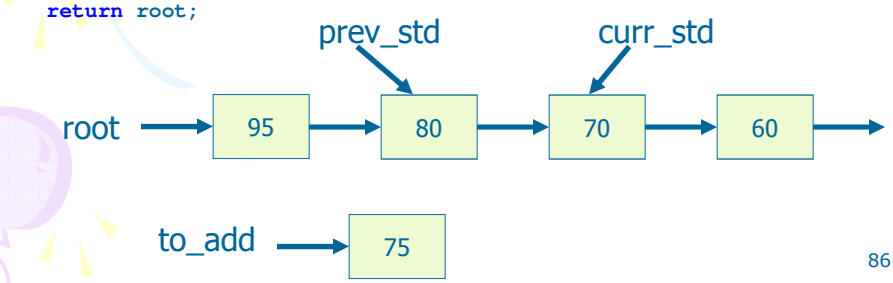


85

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;
```

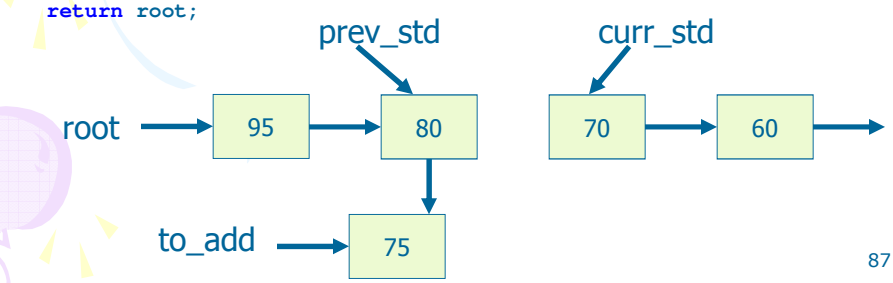


86

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;
```

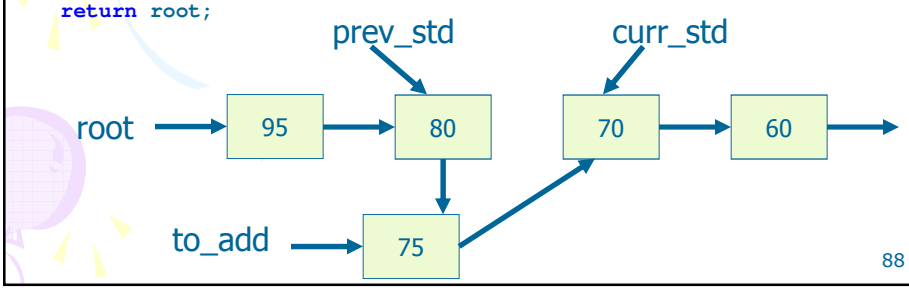


87

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;
```



88

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

89

Solution

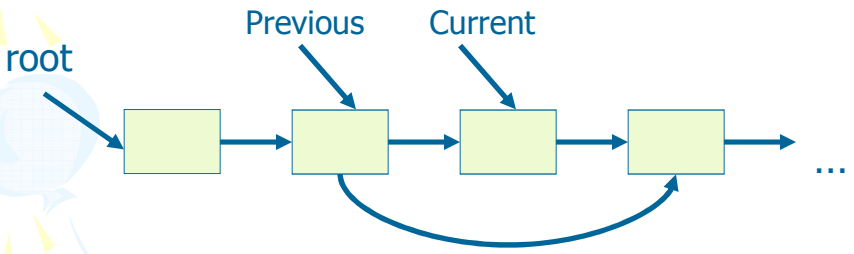
```
/* find a student whose id matches the given id number */
Student *find_student(Student *root, char* id)
{
   Student *to_search = root; /* Start from root of list */
   while (to_search != NULL) /* go over all the list */
   {
       if (strcmp(to_search->id, id) == 0) /* same id */
             return to_search;
       to_search = to_search->next;
   }
   /* If we're here, we didn't find */
   return NULL;
}
```

90

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

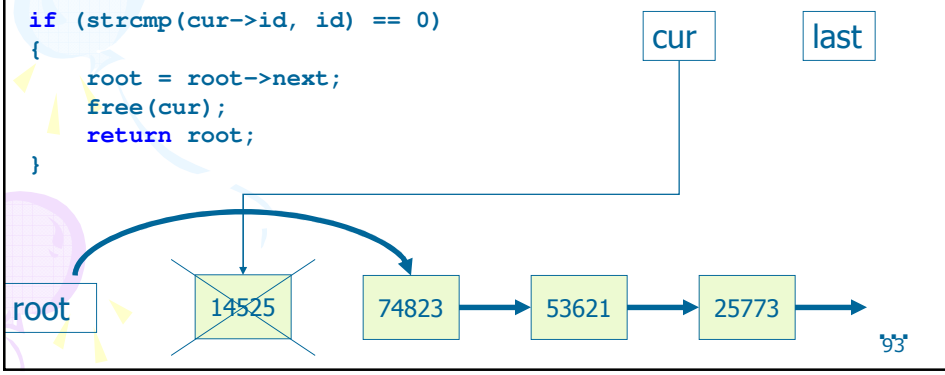
```
if (root == NULL)
   return root;

cur = root;

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

ID

14525



93

Removing a student – mid list

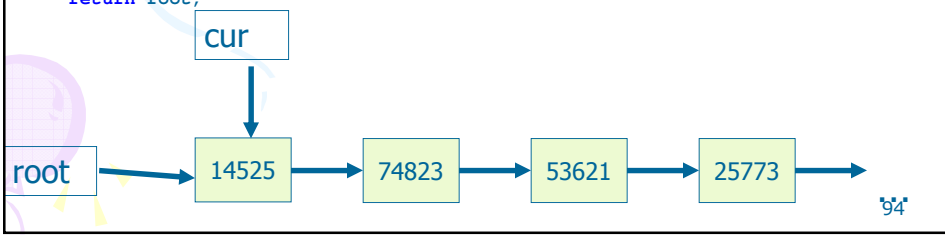
```
while (cur != NULL  &&  strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

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

return root;
```

ID

53621



94

Removing a student – mid list

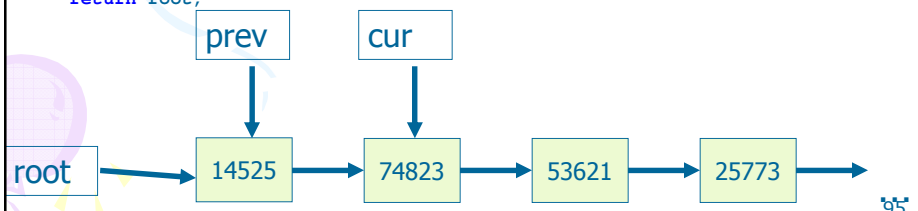
```
while (cur != NULL  &&  strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

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

return root;
```

ID

53621



95

Removing a student – mid list

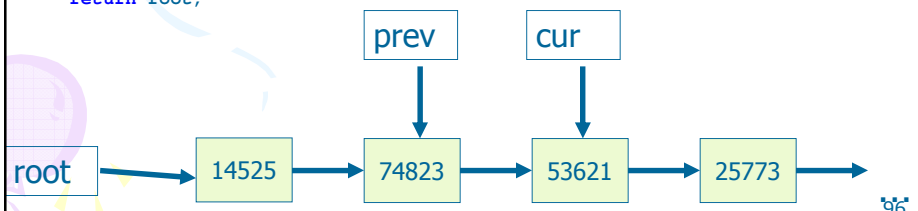
```
while (cur != NULL  &&  strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

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

return root;
```

ID

53621



96

Removing a student – mid list

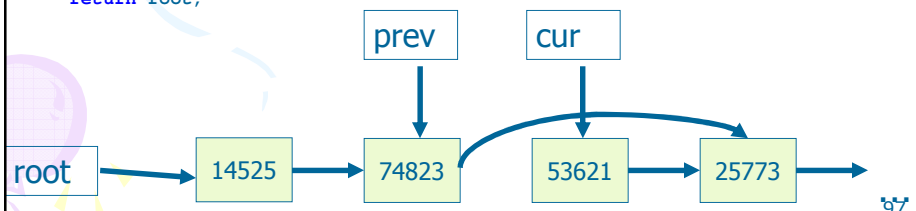
```
while (cur != NULL  &&  strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

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

return root;
```

ID

53621



97

Removing a student – mid list

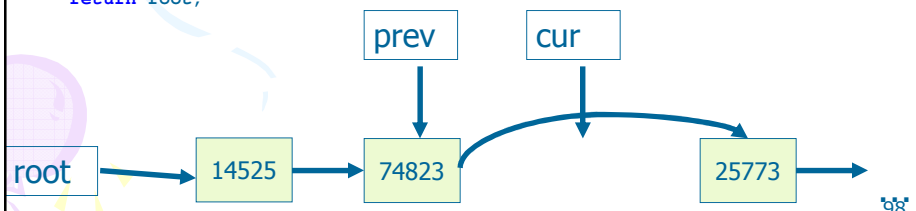
```
while (cur != NULL  &&  strcmp(cur->id, id) != 0)
{
    prev = cur;
    cur = cur->next;
}

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

return root;
```

ID

53621



98

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

99

solution

```
Student *find_student(Student* root, char* id)
{
    Student* curr = root;

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

    return curr;
}


Student *change_grade(Student *root, char* id, int new_grade)
{
    Student* std = find_student(root, id);
    std = create_student(std->name, id, new_grade);

    root = remove_student(root, id);
    return add_student(root, std);
}
```

100

Homework

- Reuse file grade.dat (output previous assignment related to transcript) as the input of STUDENT MANAGEMENT program. It has an menu-based interface.
- Apart from the functionalities already implemented on class such as Add new, delete, search and update grade, do the following tasks:
  - Print the student list in ascending order of grades.
  - Compute the average grade of the class
  - Classify grades in to 3 categories:
    - Average: The difference with the average grade is not greater than 0.5
    - Good: greater than 0.5
    - Bad: smaller than average depass 0.5
  - Store the classification to a file.

101

Homework–PhoneDB

- **Write program PhoneDB that:**
  - **Import PhoneDB.dat (previous exercise) into a single linked list and display information on screen.**
  - **Search phone by model and update information for a phone.**
  - **Display all phones in the list where the Model starts with a letter entered from the user: for example n, e (not case sensitive), sorted by the price.**
  - **Delete all phones in the list of which the model starts with a specific letter (inputted by users)**
  - **Delete n nodes (n is a number inputted by users) from a node corresponding a specific phone model.**
  - **Print out the list sorted by price**
  - **Print out a list of phones priced less than an input value**

102

Question

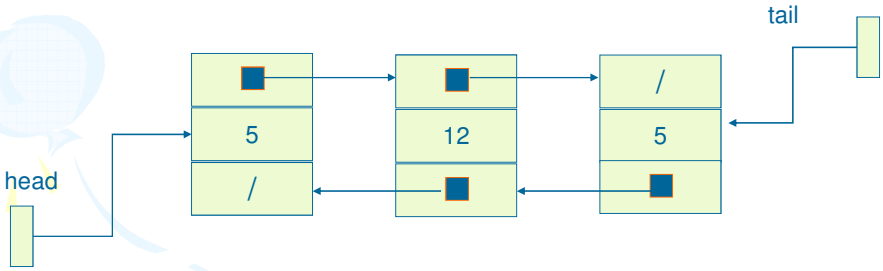
- We are now designing "address list" for mobile phones.
- You must declare a record structure that can keep a name, a phone number, and a e-mail address at least. And you must make the program which can deals with any number of the data.
- Hint: you can organize elements and data structure using following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

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

103

Double link list

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



104

Declaration

```
typedef ... elementtype;
struct node_t {
   elementtype element;
   struct node_t   *next;
   struct node_t   *prev;
};
typedef struct node_t node;
typedef node* doublelist;
doublelist head, tail, cur;
```

105

Initialisation and check for emptiness

```
void MakeNull_List (doublelist *root,
  doublist *tail, doublist *cur){
   (*root)= NULL;
   (*tail)= NULL;  (*cur)= NULL;
}
int isEmpty (doublist root){
  return (root==NULL);
}
```

106

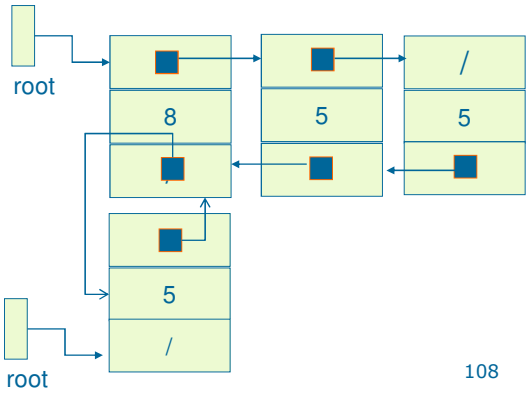
makeNewNode() - generic

```
node* makeNewNode(elementtype addr){
  node* new = (node*)
     malloc(sizeof(node));
  new->element=addr;
  new->next =NULL;
  new->prev =NULL;
  return new;
}
```

107

Insert At Head (list using global variables)

```
void insertAtHead(elementtype ele){
  node* new = makeNewNode(ele);
  if (root==NULL)      {
     root= new;
     tail= new;
     cur = new;
     return;
  }
  new->next = root;
  root->prev = new;
  root = new;
  cur = root;
}
```



108

Insert At Head (passing pointers as arguments)

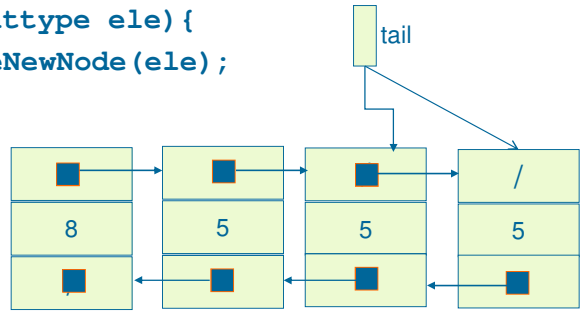
```
void insertAtHead(elementtype e, doublelist
  *root, doublelist *tail, doublelist *cur){
  node* new = makeNewNode(e);
  if (*root==NULL) {
      *root=new; *tail= new;
      *cur=new; return;
  }
  new->next = *root;
 *root->prev = new;
  *root = new;
  *cur = *root;
}
```

call in main(): insertAtHead(e, &root, &tail, &cur);

109

Insert at Tail (append-pushBack)

```
void append(elementtype ele){
  node* new = makeNewNode(ele);
  if (tail==NULL){
      root= new;
      tail= new;
      cur = new;
      return;
  }
  tail->next = new;
  new->prev = tail;
  tail = new;
  cur = tail;
}
```



tail

110

Insert at Tail (passing pointers as arguments)

```
void append(elementtype e,
  doublelist *root, doublelist
  *tail, doublelist *cur){
  node* new = makeNewNode(e);
  // fill in the code…


}
```

call in main(): append(e, &root, &tail, &cur);

111

insertion after the current position (global)

```
void insertAfterCurrent(elementtype ele){
  node *new = makeNewNode(ele);
  if(root == NULL){
       root = new; tail = new; cur = new;
  } else if (cur == NULL) {
  printf("Current pointer is NULL.\n"); return;
  } else {
    // implement your self
  ….

   //……
  }
}
```

112

insertion just after the current position (generic)

```
void insertAfterCurrent(elementtype e, doublelist
   *root, doublelist *tail, doublelist *cur) {
    // implement your self


   //….
}
```

113

insertBeforeCurrent

```
void insertBeforeCurrent(elementtype e) {
   node* new = makeNewNode(e);
    if ( root == NULL ) { /* if there is no element */
      root = new;
      tail = new;
       cur = new;
    } else {
        // Fill in the code here
        // ….


    }
}
```

114

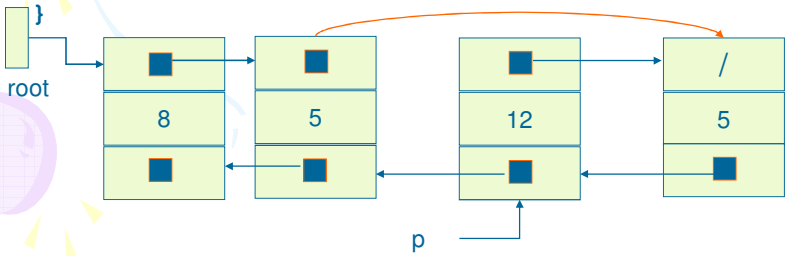
insertAtPosition

```
void insertAtPosition(elementtype e,
  int position){
  cur = root; int i;
  // implement your self by making a
  // little change from the code of
  // singly linked list
}
```

115

Delete a node pointed by p

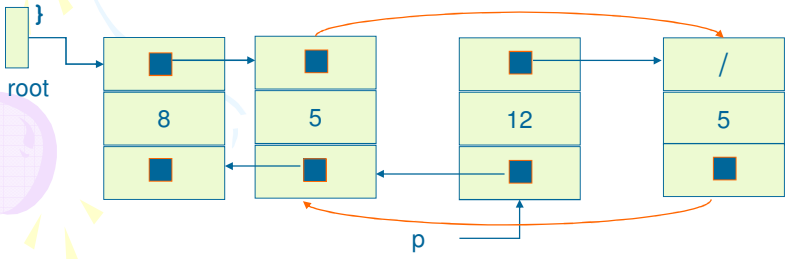
```
void Delete_List (doublelist p, doublelist *root){
 if (*root == NULL) printf("Empty list");
 else {
   if (p==*root) (*root)=(*root)->Next;
   //Delete first element
   else p->prev->next=p->next;
   if (p->next!=NULL) p->next->prev=p->prev;
   free(p);
 }
}
```



116

Delete a node pointed by p

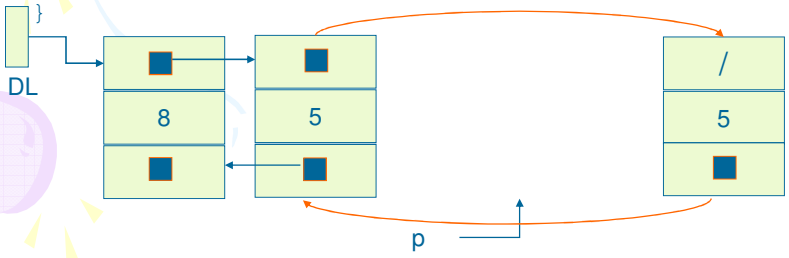
```
void Delete_List (doublelist p, doublelist *root){
 if (*root == NULL) printf("Empty list");
 else {
   if (p==*root) (*root)=(*root)->Next;
   //Delete first element
   else p->prev->next=p->next;
   if (p->next!=NULL) p->next->prev=p->prev;
   free(p);
 }
}
```



117

Delete a node pointed by p

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



118

An improvement

```
void Delete_List (doublelist p, doublelist
  *root, doublelist *cur, doublelist *tail){
 if (*root == NULL) printf("Empty list");
 else {
   if (p==*root) (*root)=(*root)->next;
   //Delete first element
   else p->prev->next=p->next;
   if (p->next!=NULL) p->next->prev=p->prev;
   else //p is tail
      *tail = p->prev;
   }
   *cur = p->prev;
   free(p);
 }
}
```

119

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

120

Summary of the functionalities of the PhoneDB exercises (Doubly Linked List)

- 1. Import from PhoneDB.dat (insertafter)
- 2. Display (choose direction: forward or backward)
- 3. Add new phone: insertbefore(1)/after(2)/atfirst(3)/append(4)
- 4. Insert at Position
- 5. Delete at Position
- 6. Delete current
- 7. Delete first
- 8. Delete last
- 9. Search and Update: (Search by model – update all field of information)
- 10. Reverse List
- 11. Save to File
- 12. Quit(Free)

121

Homework

- Build the doubly linked list library (doublylinkedlist.h) and use it in the phonebook management program.

122

Double linked list

- 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. (Bài tập)

123

Homework– Double linked list

- **Build a program with a menu interface with the following functions using the doublelinkedlist library. The program has the function of managing student information, grades - data loaded from grade.dat and imported from users**
  - **1. Import from Grade.dat (insertafter)**
  - **2. Display (traverse)**
  - **3. Add new phone (insertbefore/after)**
  - **4. Delete current**
  - **5. Delete first**
  - **6. Delete last**
  - **7. Search and Update**
  - **8. Save to File**
  - **9. Quit(Free)**

124

Doubly linked list exercises

- From file grade.dat created from previous exercises (bangdiem.txt – grade.dat,..) with the following features in menu interface.
- Initiate: Build 2 doubly linked lists. The first list contain students who have grade greater than 8 and the second list stores the rest.
- Add student: Ask data from user about a new student and add it to proper list.
- Move students: Find and move all students with grade of 7.5 from second list to first list.
- Concatenate: Append the second list at the end of the first list.
- Concatenate and sort: Merge two list to get one sorted list in the descending grade of students.

125