CS122 Algorithms and Data Structures

MW 11:00 am - 12:15 pm, MSEC 101

Instructor: Xiao Qin

Lecture 4: Linked Lists

Lists

- A list is a varying-length, linear collection of homogeneous elements.
- Linear means each list element (except the first) has a unique predecessor, and each element (except the last) has a unique successor
- In a linear structure, components can only be accessed sequentially one after the other

Lists (cont.)

- It normally has two "special" named elements called head and tail. They point to the first and last element of the list.

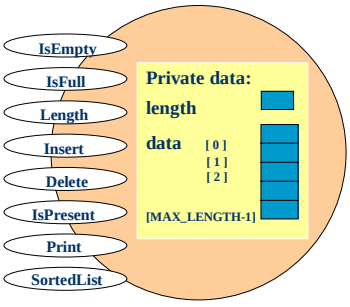
Lists as ADTs

- Domain
  - A collection elements of the same type
  - An implicit list cursor in the range 1 through *n*+1where n is the current length of the list.
- Operations
  - `create()`: create a new list. Should be done using constructors
  - `Boolean IsEmpty()` & `Boolean IsFull()`: returns true if the list is empty and full respectively
  - `void InsertBeginning(v)` & `void InsertEnd(v)`: Insert the value v either in the end or the beginning of the list
  - `void Delete()`: item at list cursor deleted
  - `print()`: Print the whole list

More List Operations

- Operations
  - `insertAfter(v, anotherV)`: insert the value v after the first occurrence of `anotherV`.
  - `insertBefore(v, anotherV)`: Insert the value v before the first occurrence of `anotherV`.
  - `search(v)`: search for the first occurrence of v in the list and return its position. Could be used to help the implementation of several other operations
  - `deleteAll(v)`: Delete all occurrences of v in the list
  - `reset()`: List cursor is at front of list.
- We can assume that private data in a list includes the lists itself (using an array or linked list), the `head`, the `tail` and the `size` of the list

Array-based `class List`



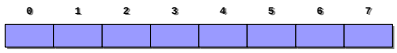
```
// ARRAY-BASED LIST    ( list.h )
const int MAX_LENGTH = 50 ;
typedef  int  ItemType ;

class  SortedList
{
public :                          // public member functions

    SortedList ( ) ;                                  // constructor
    bool        IsEmpty ( ) const ;
    bool        IsFull ( )  const ;
    int         Length ( )  const ;          // returns length of list
    void        Insert ( ItemType  item ) ;
    void        Delete ( ItemType  item ) ;
    bool        IsPresent( ItemType  item )  const ;
    void        Print ( ) ;

private :                         // private data members

    int              length ;          // number of values currently stored
    ItemType         data[MAX_LENGTH] ;
    void   BinSearch (  ItemType item, bool& found, int& position ) const ;
} ;
```

7

How to Implement a List

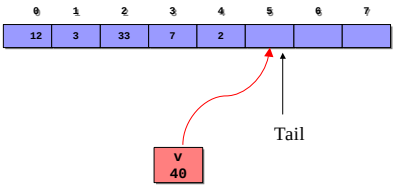
- use a built-in array stored in contiguous memory locations, implementing operations Insert and Delete by moving list items around in the array, as needed
- use a linked list (to avoid excessive data movement from insertions and deletions) not necessarily stored in contiguous memory locations

List Implementation via Arrays `create()`

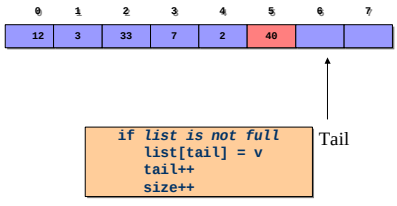



```
MAX_SIZE = 8
size = 0
head = -1
tail = 0
```

List Implementation via Arrays `insertEnd(40)`

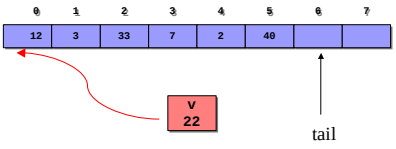


List Implementation via Arrays `insertEnd(40)`

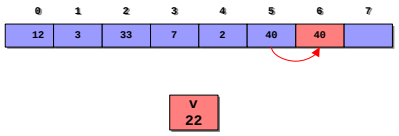



```
if list is not full
    list[tail] = v
    tail++
    size++
```

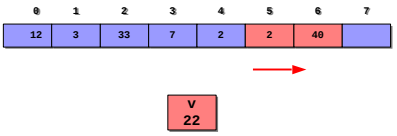
List Implementation via Arrays `insertBeginning(22)`



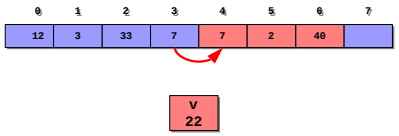
List Implementation Via Arrays
`insertBeginning(22)`



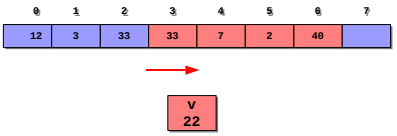
List Implementation via Arrays
`insertBeginning(22)`



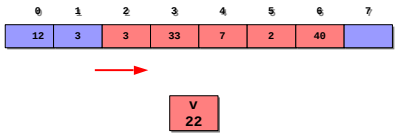
List Implementation via Arrays
`insertBeginning(22)`



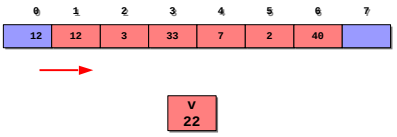
List Implementation via Arrays
`insertBeginning(22)`



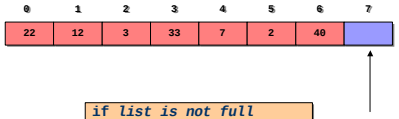
List Implementation via Arrays
`insertBeginning(22)`



List Implementation via Arrays
`insertBeginning(22)`

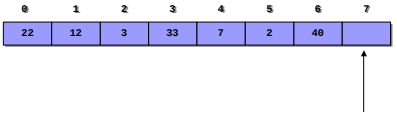


List Implementation via Arrays
`insertBeginning(22)`

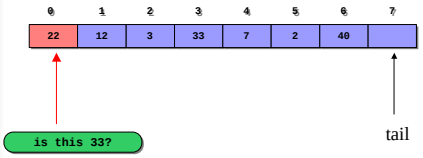



```
if list is not full
   for i=(tail-1) to 0
      list[i+1] = list[i]
   list[0] = v
   tail++
   size++
```

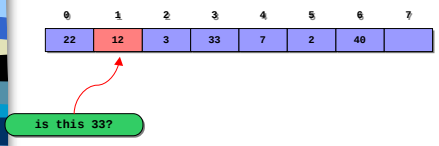
List Implementation via Arrays
`delete(33)`



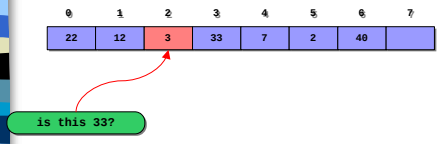
List Implementation via Arrays
`delete(33)`



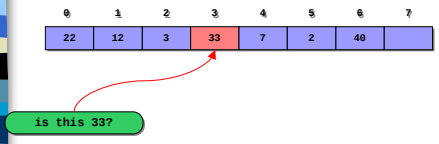
List Implementation via Arrays
`delete(33)`



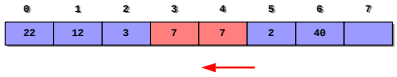
List Implementation via Arrays
`delete(33)`



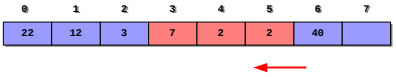
List Implementation via Arrays
`delete(33)`



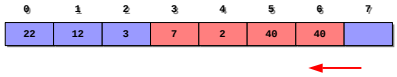
List Implementation via Arrays
`delete(33)`



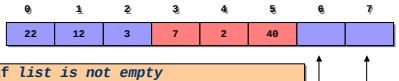
List Implementation via Arrays
`delete(33)`



List Implementation via Arrays
`delete(33)`



List Implementation via Arrays
`delete(33)`




```
if list is not empty
   i = head+1;
   while (v != list[i] and i < tail)
      i++
   if (i < tail)
      for j=i to (tail-2)
         list[i] = list[i+1]
      list[tail-1] = null;
      size--
      tail--
```

tail' tail

How to Implement a List

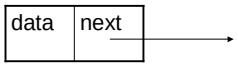
- use a built-in array stored in contiguous memory locations, implementing operations Insert and Delete by moving list items around in the array, as needed
- use a linked list (to avoid excessive data movement from insertions and deletions) not necessarily stored in contiguous memory locations

Linked Lists vs. Arrays

- Lists differ from arrays because they don't have a fixed size but like arrays can only store elements of the same type
- Main advantage over arrays is easy insertion and deletion of nodes
- A well defined list may be the basis for the implementation of several other data structures, such as queues, stacks, trees and graphs.

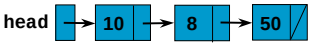
Self-referential data types

```
class Node {
   private Object info;     // the "info"
   private Node next;       // the "link"
}
```



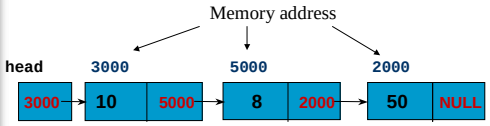
A Linked List

- a linked list is a list in which the order of the components is determined by an explicit link member in each node
- the nodes are structs--each node contains a component member and also a link member that gives the location of the next node in the list
- an external pointer (or head pointer) points to the first node in the list



Nodes can be located anywhere in memory

the link member holds the memory address of the next node in the list

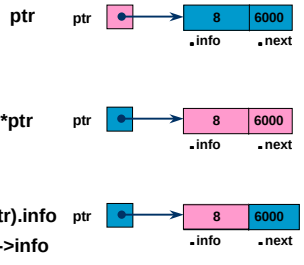


Declarations for a Linked List

```
// Type DECLARATIONS
struct NodeType  {
  int         info;
  NodeType*   next;
}

// Variable DECLARATIONS
NodeType* head;
NodeType* ptr;
```

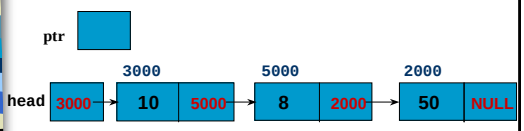
34

Member Selection



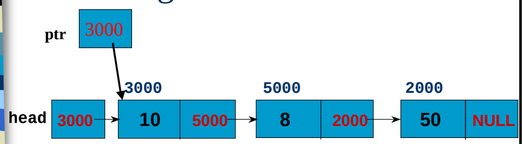
35

Traversing a Linked List



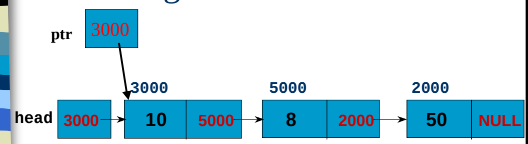

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
       cout  <<  ptr->info ;
                    // Or, do something else with
       ptr  =  ptr->next ;
}
```

Traversing a Linked List



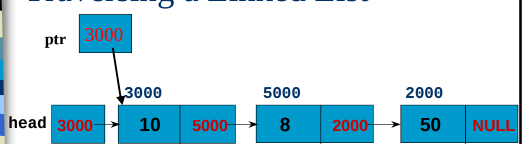

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                     // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



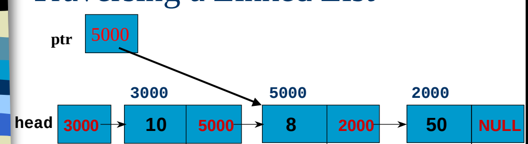

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                     // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



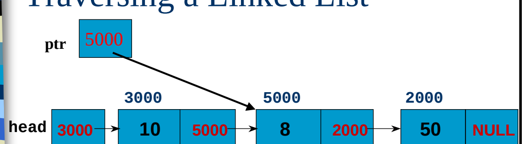

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                     // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



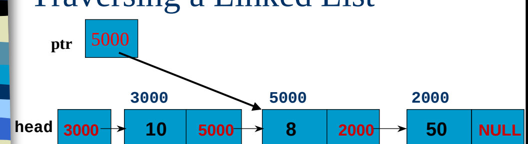

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                     // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



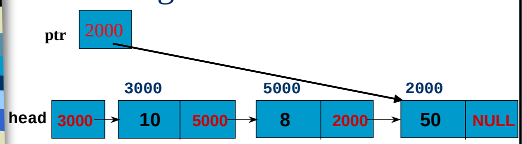

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                     // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



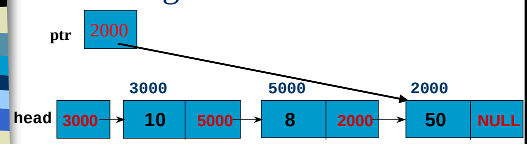

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                     // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



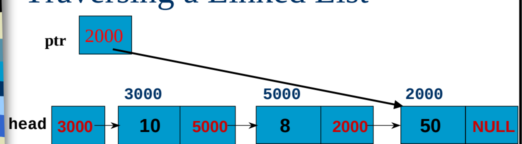

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



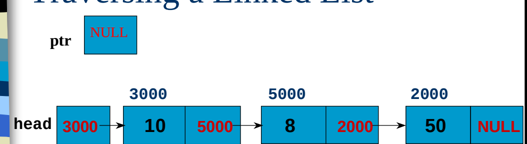

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List



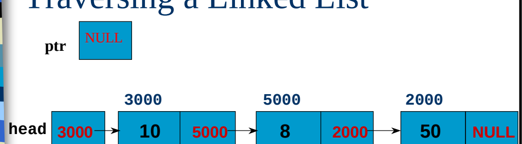

```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List




```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                // Or, do something else with
        ptr  =  ptr->next ;
}
```

Traversing a Linked List




```
//PRE:  head points to a linked list
node *ptr ;

ptr  =  head ;
while (ptr != NULL)  {
        cout  <<  ptr->info ;
                // Or, do something else with
        ptr  =  ptr->next ;
}
```

Using Operator `new`

If memory is available in an area called the free store (or heap), operator new allocates the requested object, and returns a pointer to the memory allocated.

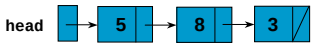
The dynamically allocated object exists until the delete operator destroys it.

48

Inserting a Node at the Front of a List

item 6

```
int     item = 6;
Node   *location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```

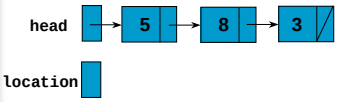


49

Inserting a Node at the Front of a List

item 6

```
int     item = 6;
Node   *location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```

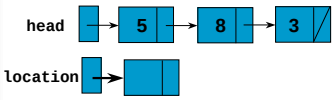


50

Inserting a Node at the Front of a List

item 6

```
int     item = 6;
Node   *location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```

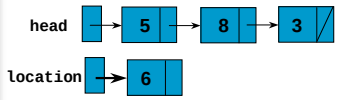


51

Inserting a Node at the Front of a List

item 6

```
int     item = 6;
Node   *location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```

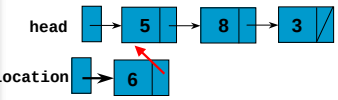


52

Inserting a Node at the Front of a List

item 6

```
int     item = 6;
Node   *location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```

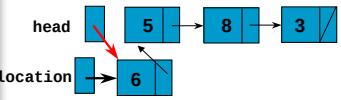


53

Inserting a Node at the Front of a List

item 6

```
int     item = 6;
Node   *location;
location = new  NodeType;
location->info = item;
location->next = head;
head = location;
```



54

Using Operator `delete`

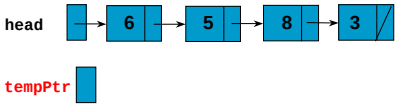
- The object currently pointed to by the pointer is deallocated, and the pointer is considered undefined. The object's memory is returned to the free store.

55

Deleting the First Node from the List

item

```
Node    *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
```

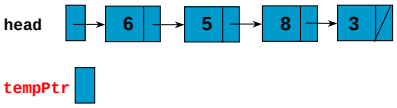


56

Deleting the First Node from the List

item 6

```
Node    *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
```

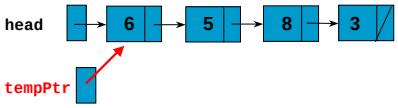


57

Deleting the First Node from the List

item 6

```
Node    *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
```

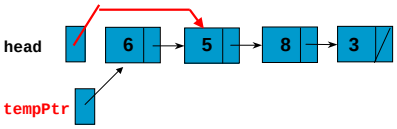


58

Deleting the First Node from the List

item 6

```
Node    *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
```



59

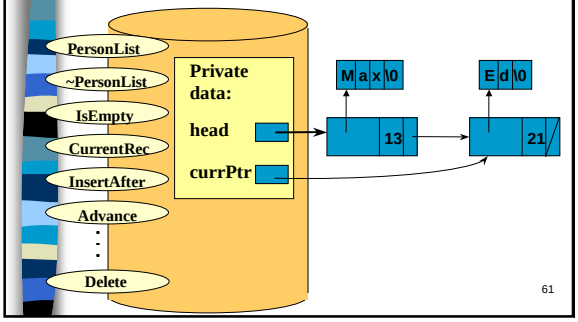
Deleting the First Node from the List

item 6

```
Node    *tempPtr;

item = head->info;
tempPtr = head;
head = head->next;
delete  tempPtr;
```


60

class PersonList



61

```
// SPECIFICATION FILE                person.h

#include  "bool.h"
   . . .

struct PersonRec
{
    char*   name ;        // Pointer to person's name
    int     age ;         // Person's age
} ;

struct  PersonNode
{
    char*       name ;      // Pointer to person's name
    int         age ;       // Person's age
    PersonNode *next ;      // Pointer to next node in list
} ;
```

62

```
// SPECIFICATION FILE    continued          person.h

class  PersonList
{
public : //  LINKED LIST IMPLEMENTATION
    PersonList ( ) ;
    ~PersonList ( ) ;
    Boolean    IsEmpty ( ) const ;
    Boolean    IsFull ( )  const ;
    void       Reset ( ) ;
    Boolean    EndOfList ( ) const ;
    void       InsertAfter ( PersonRec  someRec ) ;
    void       InsertBefore ( PersonRec someRec ) ;
    void       Advance ( );
    void       Delete ( ) ;
    PersonRec CurrentRec ( ) const ;

private :
    PersonNode  *head ;
    PersonNode  *currPtr ;
} ;
```

63

```
// LINKED LIST IMPLEMENTATION FILE   ( person.cpp )
#include "person.h"
 . . .

PersonList::PersonList ( )      // constructor
//  PRE:    None.
// POST:   Empty list created && EndOfList( ).
{
    head = NULL;
    currPtr = NULL;
}

Boolean  PersonList::IsEmpty ( )  const
// POST:  FCTVAL == ( list is empty )
{
    return  ( head == NULL ) ;
}
```

64

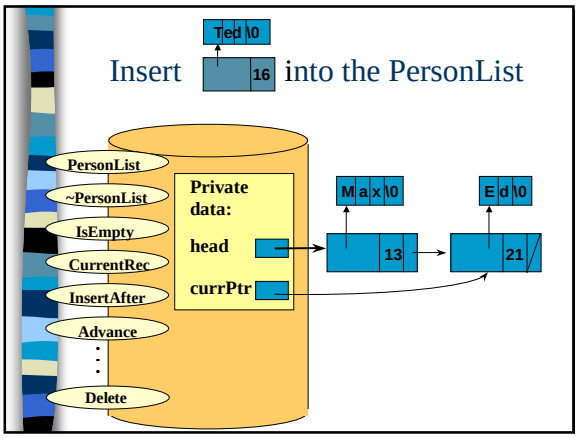
```
// IMPLEMENTATION  CONTINUED          ( person.cpp )

void  PersonList::Reset ( )
//  PRE:    NOT  IsEmpty ( )
// POST:   List cursor is at front of list
{
    currPtr = head  ;
}

PersonRec  PersonList::CurrentRec ( )  const
//  PRE:    NOT  IsEmpty ( )  && NOT  EndOfList ( )
// POST:  FCTVAL == record at list cursor
{
    PersonRec   rec ;

    rec.name = currPtr->name ;
    rec.age = currPtr->age ;
    return  rec ;
}
```

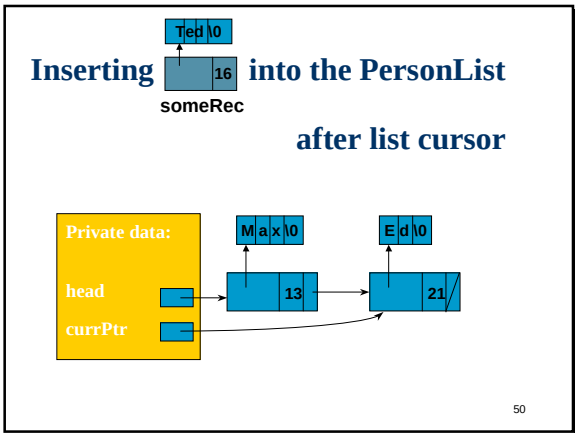
65

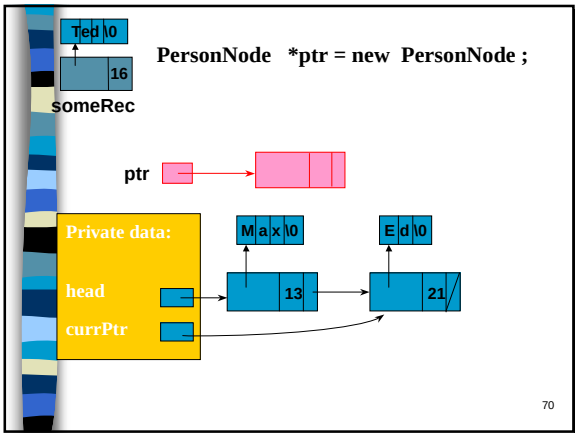
```
Boolean  PersonList::EndOfList ( )  const
// POST: FCTVAL == ( list cursor is beyond end of list )
{
    return  ( currPtr == NULL ) ;
}

void  PersonList::Advance ( )
//  PRE:  NOT  IsEmpty( )  && NOT  EndOfList( )
// POST:   List cursor has advanced to next record
{
    currPtr = currPtr->next  ;
}

PersonList::~PersonList ( )     // destructor
// POST:   List destroyed
{
    currPtr = head ;
    while  ( ! EndOfList ( ) )
        Delete ( ) ;
}
```

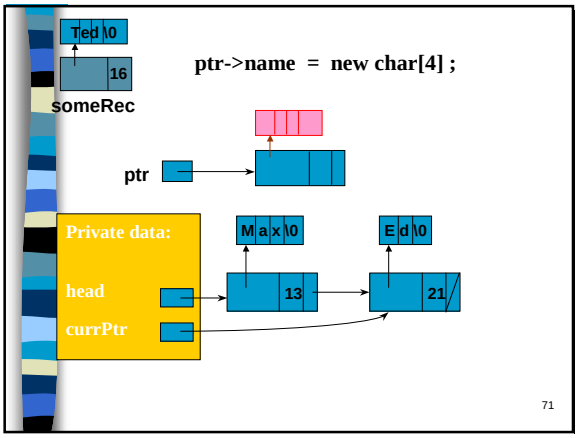
66

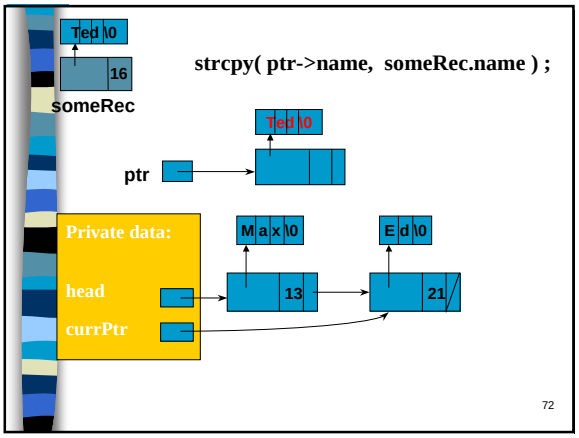
```
void PersonList::InsertAfter ( PersonRec  someRec )

// PRE:  Assigned (someRec)  && NOT  IsEmpty( )
//           && NOT  IsFull ( )  &&  NOT  EndOfList ( )

// POST:   someRec inserted after list cursor
//             &&  This new node has become the current record
{
                                        // obtain and fill a node
    PersonNode  *ptr  = new  PersonNode ;

    ptr->name = new  char [ strlen ( someRec.name) + 1 ] ;
    strcpy( ptr->name, someRec.name );
    ptr->age = someRec.age ;
    ptr->next = currPtr->next ;
    currPtr->next = ptr ;
    currPtr  = ptr ;
}
```

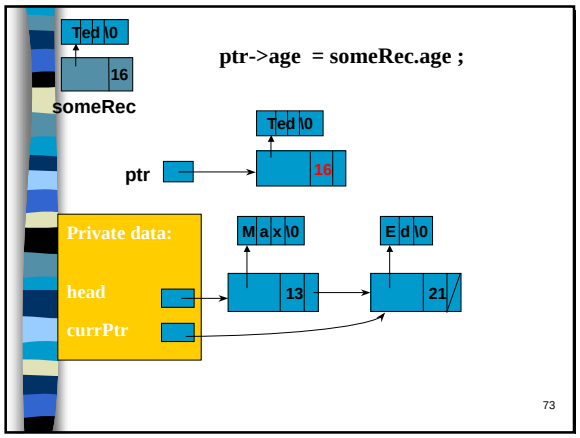


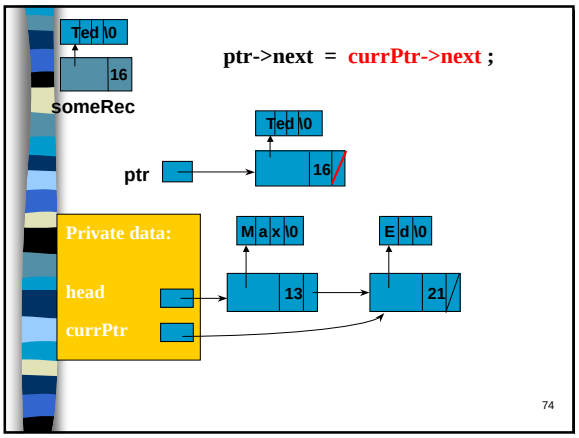


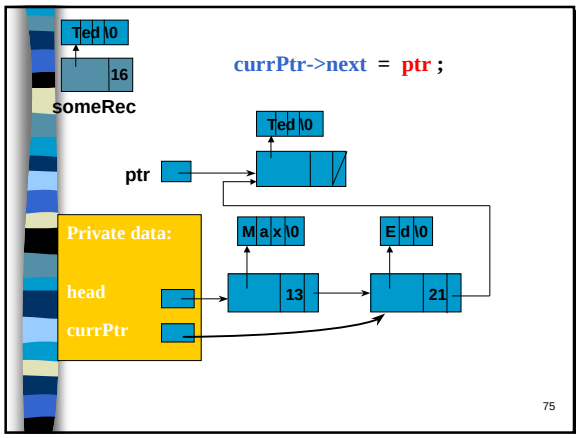


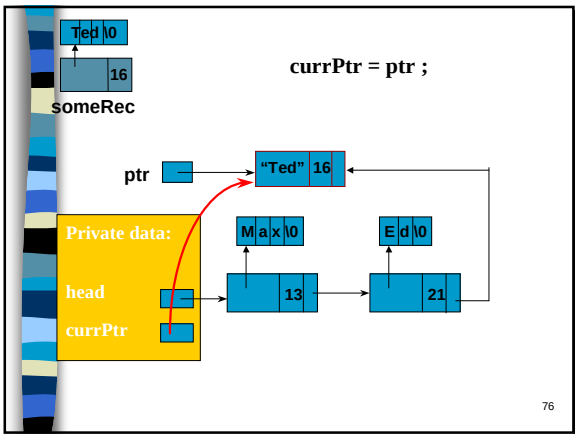





Ted \0
16
someRec
ptr = someRec.age ;
Ted \0
16
ptr
Private data:
head
currPtr
M a x \0
13
E d \0
21
73


Ted \0
16
someRec
ptr->next = currPtr->next ;
Ted \0
16
ptr
Private data:
head
currPtr
M a x \0
13
E d \0
21
74


Ted \0
16
someRec
currPtr->next = ptr ;
Ted \0
ptr
Private data:
head
currPtr
M a x \0
13
E d \0
21
75


Ted \0
16
someRec
currPtr = ptr ;
ptr
"Ted" 16
Private data:
head
currPtr
M a x \0
13
E d \0
21
76