

Linked Lists

Table of Contents

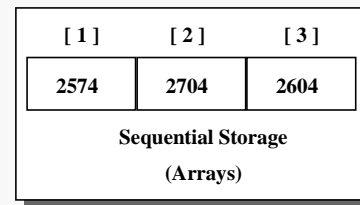
- Linear Lists
- Self Referencing Structures
- Linked List Operations
- Elementary Operations
- List Insertion Cases
- Ordered Insertion
- List Remove Cases
- Ordered Remove
- Merge Lists (no preservation)
- Merge Lists (preservation)
- Other List Functions
- Linear Linked-List Variations
- Linked-List Variation Declarations
- Circular Double Linked Insertion
- Circular Double Linked Deletion

- Linked-List Class Declarations
- Linked-List Class Definitions
- Linked-List :: modify
- Linked-List :: insert
- Linked-List :: remove

Linear Lists

Sequential Lists

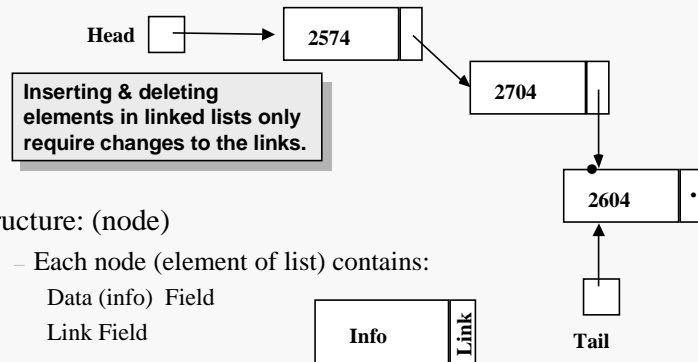
- Array
- Logical order matches physical order
- i.e.: Class List = (2574, 2704, 2604)



Inserting & deleting elements in sequential lists requires copying & shifting of elements.

Linked Lists

- Logical group of ordered elements whose physical order is independent of physical storage



Self Referencing Structures

5. Linked Lists 3

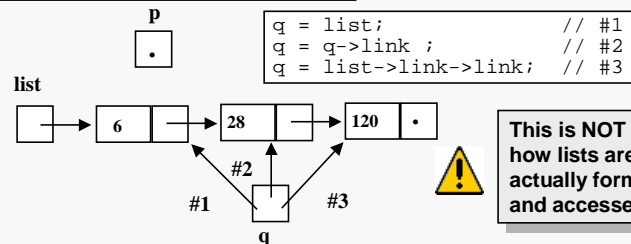
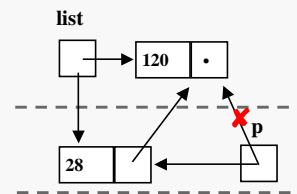
Linked Lists Declaration

```
struct node; //forward declaration
typedef node* nodePtr;
typedef int infoType;
struct node {
    infoType element;
    nodePtr link;
};
nodePtr list = NULL; //head ptr
```

C provides for the declaration of pointers to structures before their definition.

List Formation Example

```
nodePtr p = NULL, q = NULL;
// assuming a list of ints
p = new node;
p->element = 120;
p->link = NULL;
list = p;
-----
p = new node;
p->element = 28;
p->link = list;
list = p;
-----
p = new node;
p->element = 6;
p->link = list;
list = p;
//delete p; ???
p = NULL;
```



Linked List Operations

5. Linked Lists 4

Ordered List Functions

createList(L) : initializes L to be empty

emptyList(L) : tests if L is empty

fullList(L) : tests if L is full

retrieveElem(L, key, elem, found) :
returns element with key value Key
if it exists in L

insert(L,newElem) : inserts newElem in L

remove(L, delElem) : deletes element with same key
value as delElem

modify(L,modElem) : replaces existing element with
same key value as modElem

printList(L) : prints all the elements in L

sizeList(L) : returns the number of elements in
the list

destroyList(L) : deletes all elements in L

// L is the head pointer.

Linked List Operations

5. Linked Lists 5

Considerations

- Element (elem, newElem, delElem, modElem) is of type infoType
- key is the member field of infoType that the list is ordered upon

Type Defined Declaration

```
struct node; //forward declaration
typedef node* nodePtr;

typedef struct {
    infoType element;
    nodePtr link;
} node;

nodePtr list = NULL ;
```

Localize dependencies:

```
bool lessThan(node elem1, node elem2);
bool equalTo(node elem1, node elem2);
bool greaterThan(node elem1, node elem2);
```

Elementary Operations

5. Linked Lists 6

Initialization

```
//PreCond: L is undefined
void createList ( nodePtr& L) {
    L = NULL; //set L = empty list
}
```

Empty Test

```
bool emptyList ( nodePtr L) {
    return( L == NULL );
}
```

**//alternatively implement one line
//operations as inline functions**

Full Test

// Array Linked List Implementation.

printList

```
void printList ( nodePtr L) {
    nodePtr p = NULL;
    for (p = L; p != NULL; p = p->link)
        printElem(p->element);
}
```

**List
Traversal**

sizeList

```
int sizeList ( nodePtr L) {
    nodePtr p = NULL;
    int x = 0;
    for (p = L; p ; p = p->link) x++;
    return x;
}
```

List Insertion Cases

5. Linked Lists 7

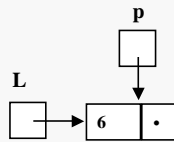
Declarations:

```
nodePtr L = NULL;
nodePtr p = NULL;
```

Four Cases

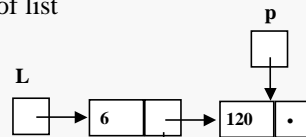
– Case #1: Insert into empty list

```
p = new node;
p->element = 6;
p->link = NULL;
L = p;
```



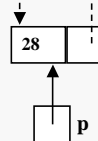
– Case #2: Insert at end (tail) of list

```
p = new node;
p->element = 120;
p->link = NULL;
L->link = p;
```



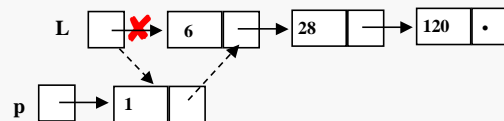
– Case #3: Insert into middle of the list

```
p = new node;
p->element = 28;
p->link = L->link;
L->link = p;
```



– Case #4: Insert at the front (head) of the list

```
p = new node;
p->element = 1;
p->link = L;
L = p;
```



Ordered Insertion

5. Linked Lists 8

Ordered (ascending) List Insertion Function

```
bool insert( nodePtr& list, infoType elem) {
    nodePtr prevPtr, currPtr, newPtr;
    newPtr = new node;

    if (newPtr == NULL)
        return false;           // heap is empty

    newPtr->element = elem;
    newPtr->link = NULL;

    prevPtr = NULL;
    currPtr = list;

    while ((currPtr != NULL) &&
           (greaterThan(elem, currPtr->element))) {
        prevPtr = currPtr;
        currPtr = currPtr->link;
    }

    if (prevPtr == NULL) {      //insert at head, if
        newPtr->link = list;    // empty list
        list = newPtr;
    }
    else {
        prevPtr->link = newPtr; //insert in middle
        newPtr->link = currPtr; // or at tail
    }
    return true ;              // successful insertion
}
```

prevPtr is used as a 'trailer' pointer

depends on Boolean short-circuiting

head of list pointer (list) must be passed by reference.

lessThan, equalTo, greaterThan Boolean functions are coded specific to list element type.

List Remove Cases

5. Linked Lists 9

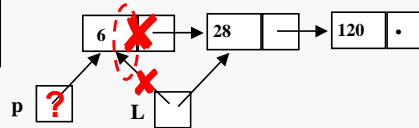
Consider the list:



Three Cases

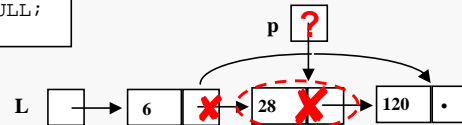
– Case #1: Remove the head of the list

```
p = L;
L = L->link;
//p->link = NULL;
delete p;
```



– Case #2: Remove from middle of list

```
p = L->link;
L->link = p->link;
//p->link = NULL;
delete p;
```



– Case #3: Remove last element in list

Identical to removal of head.

Ordered Remove

5. Linked Lists10

Ordered (ascending) List Removal Function

```
bool remove( nodePtr& list, infoType delElem) {
// one-element node lookahead
nodePtr ptr, delPtr;

ptr = list;

if (emptyList(list))
return false; // removal failure

if (equalTo(delElem, list->element) {
list = list->link; // delete head
delete ptr;
return true; // successful removal
}
// check for 1-element list
if (ptr->link == NULL)
return false;

// list has > 1-element
// perform 1-element look-ahead search
while( (ptr->link->link != NULL) &&
(!equalTo(delElem, ptr->link->element))
ptr = ptr->link;

// remove middle or tail node
if (equalTo(delElem, ptr->link->element)){
delPtr = ptr->link;
ptr->link = ptr->link->link;
delete delPtr;
return true; // successful removal
}

//end of list && delElem !found
return false; // removal failure
}
```

Trailer pointer method
is also applicable

lessThan, equalTo, greaterThan
functions provide semi-generic
list operations.

Merge Lists (no preservation) *in situ*

5. Linked Lists11

```
/* Given 2 ascending ordered single linked-lists,
return a new ordered list which contains all of the
elements of both lists, (the original lists may be
destroyed by the merging). */

nodePtr mergelists( nodePtr& list1, nodePtr& list2) {
    nodePtr head, trail1, trail2;
    if (list1 == NULL) return list2;
    if (list2 == NULL) return list1;
    // set merge list head
    head = (lessThan(list1->element , list2->element)
           ? list1 : list2);

    while ( (list1 != NULL) && (list2 != NULL) ) {
        if (equalTo(list1->element, list2->element)){
            trail2 = list2 -> link; //move list2 up 1 node
            list2->link = list1; //due to possibly initial
            list2 = trail2; //equal elements
        }
        else
            if (lessThan(list1->element, list2->element) ) {
                while ((list1 != NULL) &&
                       (lessThan(list1->element, list2->element))){
                    trail1 = list1;
                    list1 = list1 -> link;
                } //while
                trail1->link = list2;
            } //if
            else { //(lessThan(list2->element,list1->element)
                while ( (list2 != NULL) &&
                       (lessThan(list2->element , list1->element)) ) {
                    trail2 = list2;
                    list2 = list2 -> link;
                } //while
                trail2->link = list1;
            } //else
        } //while
    list1 = list2 = NULL;
    return head;
}
```

Assumes elements within list are unique.

while conditions rely upon Boolean short-circuiting.

Duplicated code should be eliminated.

Problem: if List2 contains multiple items equal to head of list1?

Merge Lists (preservation)

5. Linked Lists12

```
/* Given 2 ascending ordered single linked-lists,
return a new ordered list which contains all of
the elements of both lists, (the original lists
must NOT be destroyed by the merging). */

nodePtr mergelists2( nodePtr list1, nodePtr list2){
    nodePtr merge, ptr;
    createList(merge);

    ptr = list1;
    while (!emptyList(ptr) {
        insertList(merge, ptr->element);
        ptr = ptr -> link;
    }

    ptr = list2;
    while (!emptyList(ptr) {
        insertList(merge, ptr->element);
        ptr = ptr -> link;
    }

    return merge;
}

void insertList( nodePtr ilist, nodePtr& clist)
{
    nodePtr ptr = ilist;

    while (!emptyList(ptr) {
        insert(clist, ptr->element);
        ptr = ptr -> link;
    }
}
```

Other List Functions

5. Linked Lists13

destroyList

```
void destroyList( nodePtr& list)
{
    nodePtr p = list;

    while (list != NULL) {
        list = list->link;
        delete p;
        p = list;
    }
}
```

modify

```
bool modify( nodePtr& list, infoType modElem)
{
    nodePtr p = list;
    bool foundElement= false;
    bool endList = (list == NULL);

    if (!endList)
        foundElement = equalTo(p->element, modElem);

    while (!endList && !foundElement) {
        p = p->link;
        endList= p ==NULL;
        if (!endList)
            foundElement = equalTo(p->element, modElem);
    }//while

    if (foundElement) { //replace list element
        p->element = modElem;
        return true;    //successful modification
    }

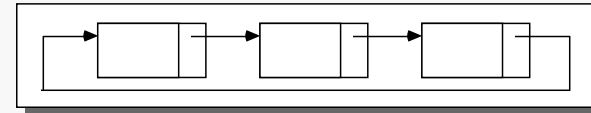
    return false;    //unsuccessful modification
}
```

**modify does NOT
rely upon Boolean
short-circuiting.**

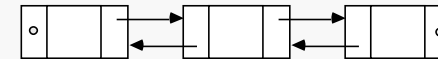
Linear Linked-List Variations

5. Linked Lists14

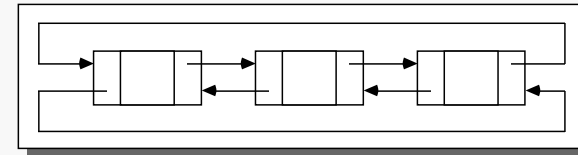
Circular List



Double Linked-List (non-circular)



Circular Double Linked-List



Linked-List Variation Declarations

5. Linked Lists15

Double Linked-List Declaration (non-circular)

```
typedef struct node *nodePtr;
typedef struct {
    infoType element;
    nodePtr prev, next;
} node;

typedef struct {
    nodePtr first, last;
} dblList;

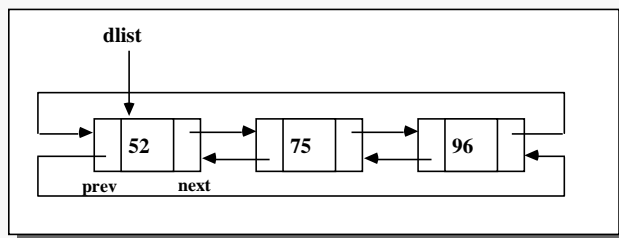
dblList dlist;
```

Some compilers
may require
forward
declaration.

Double Linked-List Declaration (circular)

```
typedef struct node *dblList;
typedef struct {
    infoType element;
    dblList prev, next;
} node;

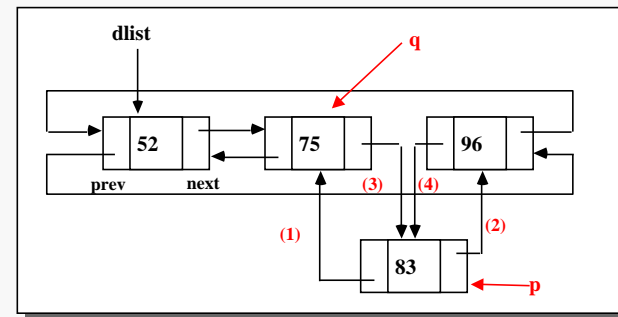
dblList dlist;
```



Circular Double Linked-List Insertion

5. Linked Lists16

Insert 83 into the ordered list:



code:

```
typedef struct node *dblList;
typedef struct {
    infoType element;
    dblList prev, next;
} node;

dblList dlist;

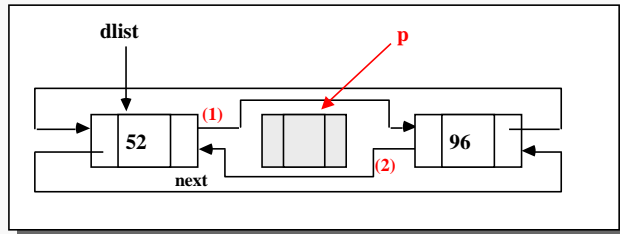
p->prev = q; // 1
p->next = q->next; // 2
q->next = p; // 3
p->next->prev = p; // 4
```

Order is important!

Circular Double Linked-List Deletion

5. Linked Lists17

Delete 75 from the list:



code:

```
p->prev->next = p->next; // 1
p->next->prev = p->prev; // 2
delete p; // 3
```

deleting the head:

```
if (dlist == p)
    dlist = p->prev;

p->prev->next = p->next; // 1
p->next->prev = p->prev; // 2
delete p; // 3
```

What if list has one element?

Merge Double Lists (no preservation)

5. Linked Lists18

```
/* Given 2 ordered circular double linked-lists, return a new
ordered list which contains all the elements of both lists,
(the original lists are destroyed */
dbList mergelists( dbList& list1, dbList& list2) {
    dbList head, trail1, trail2, last;
    if (emptyList(list1)) return list2;
    if (emptyList(list2)) return list1;

    list1->prev->next = NULL; list1->prev = NULL;
    list2->prev->next = NULL; list2->prev = NULL;
    head = (lessThan(list1->element , list2->element)
            ? list1 : list2; //set merge list head
    while ((list1 != NULL) && (list2 != NULL) ) {\
        if (equalTo(list1->element , list2->element) ) {
            trail2 = list2 -> next; //move list2 up 1 node
            list2->next = list1; // due to possibly initial
            list2 = trail2; // equal elements
        } else
            if (lessThan(list1->element , list2->element) ) {
                while ((list1 != NULL) &&
                    (lessThan(list1->element , list2->element) )){
                    trail1 = list1;
                    list1 = list1 -> next;
                } //while
                trail1->next = list2; list2->prev = trail1;
            } //if
            else { //((lessThan(list2->element , list1->element)
                while ((list2 != NULL) &&
                    (lessThan(list2->element , list1->element) )){
                    trail2 = list2;
                    list2 = list2 -> next;
                } //while
                trail2->next = list1; list1->prev = trail2;
            } //else
        } //while
    }
    last = (list1==NULL) ? list2 : list 1; //nonempty list?
    while (last->next != NULL)
        last = last->next;
    last->next = head;
    head->prev = last;
    list1 = list2 = NULL;
    return head;
}
```

Linked-List Class Declarations 5. Linked Lists19

Header File

```
//Llist.h
#ifndef LLIST_H
#define LLIST_H

#include <iostream.h> //definition of NULL
#include "infoType.h" //client supplied list elemType
//& relational functions

struct node; //forward declaration
typedef node* nodePtr;
typedef elemType infoType;
struct node {
    infoType element;
    nodePtr link;
};

class List {
public:
    List(); //constructor aka createList()
    ~List(); //destructor aka destroyList()

    bool emptyList() const; //tests if L is empty

    bool retrieveElem(infoType key, infoType& elem);
    //returns element with key value if it exists

    bool insert(infoType newElem); //inserts newElem

    bool remove(infoType delElem);
    //deletes element with same key value as delElem

    bool modify(infoType modElem);
    //replaces element with same key value as modElem

    void destroyList(); //deletes all elements

private:
    nodePtr list; //head ptr
};
#endif
```

Linked-List Class Definitions 5. Linked Lists20

cpp file

```
//Llist.cpp
#include "Llist.h" //class declarations

List::List(){ //constructor aka createList()
    list = NULL ;
}

List::~List(){ //destructor aka destroyList()
    this -> destroyList();
}

bool List::emptyList() const{ //tests if L is empty
    return (!list) ;
}

void List::destroyList () { //deletes all elements
    nodePtr p = list;

    while (list) {
        list = list->link;
        delete p;
        p = list;
    }
    list = NULL;
}
```

this //default object pointer

this pointer

All objects contain a language defined default pointer to themselves, termed the `this` pointer.

The `this` pointer is automatically passed with an object.

Can be “explicitly” used to reference object data and function members. (Accesses to object members automatically “implicitly” use the `this` pointer.)

Member function definitions

```

bool List:: modify(infoType modElem)
{//replaces element with same key value as modElem

  nodePtr p = (List::list); :: scope resolution operator

  bool foundElement= false;
  bool endList = (!(List::list));

  if (!endList)
    foundElement = equalTo(p->element, modElem);

  while (!endList && !foundElement) {
    p = p->link;
    endList= (!p);
    if (!endList)
      foundElement = equalTo(p->element, modElem);
  }//while

  if (foundElement) { //replace list element
    p->element = modElem;
    return true;      //successful modification
  }

  return false;      //unsuccessful modification
}

```

class::data member

The scope resolution can be used to fully qualify access to object data and function members.

Unnecessary unless local automatic identifier masks access to object member.

Insert member function definition

```

bool List:: insert(infoType elem) {

  nodePtr currPtr = list;
  nodePtr newPtr = new node; Performs one-node lookahead

  if (!newPtr)
    return false;          // heap is empty

  newPtr->element = elem; Relies upon Boolean short-circuiting
  newPtr->link = NULL;

  if ((! currPtr ) || //insert emptylist or head
      (! (greaterThan(elem, currPtr->element))) ){
    newPtr->link = list;
    list = newPtr;
    return true ; //successful insertion
  } //endif

  while ((currPtr->link) &&
        (greaterThan(elem, currPtr->link->element)))
    currPtr = currPtr->link;

  //insert in middle or at tail
  newPtr ->link = currPtr->link;
  currPtr->link = newPtr;
  return true ;          // successful insertion
}

```

Remove member function definition

```
bool List::remove(infoType delElem) {
    nodePtr prevPtr, currPtr;

    currPtr=list;
    prevPtr=NULL;

    //use trailing pointer
    while( (currPtr) &&
           (!equalTo(delElem, currPtr ->element)) ) {
        prevPtr = currPtr;
        currPtr = currPtr ->link;
    }//end while

    if (!currPtr) //list empty or not found
        return false;
    else if (!prevPtr) {
        list = list->link;    // delete head
        delete currPtr;
    }
    else { //remove middle or tail node
        prevPtr->link = currPtr->link;
        delete currPtr;
    }

    return true;           // successful removal
}
```

Relies upon Boolean short-circuiting