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

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

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

Structure: (node)
- Each node (element of list) contains:
  - Data (info) Field
  - Link Field

Linear Lists

Sequential Storage
(Arrays)

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

Link

Inserting & deleting elements in linked lists only require changes to the links.

Info

2574
2704
2604
Linked Lists

Self Referencing Structures

Linked Lists Declaration

```c
struct node; // Forward declaration
typedef node* nodePtr;
struct node {
    infoType element;
    nodePtr link;
};
nodePtr list = NULL;
```

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

List Formation Example

```c
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;
```

Linked List Operations

```
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
```

This is NOT how lists are actually formed and accessed.
Linked List Operations

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;
```

Elementary Operations

Initialization

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

Empty Test

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

Full Test

```
// Array Linked List Implementation.
void printList ( nodePtr L) {
  for (p = L; p != NULL; p = p->link)
    printElem(p->element);
}
```

List Traversal

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

Localize dependencies:

bool lessThan(node elem1, node elem2);
bool equalTo(node elem1, node elem2);
bool greaterThan(node elem1, node elem2);
List Insertion Cases

 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

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

 prevPtr is used as a 'trailer' pointer

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

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

 **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.**
Consider the list:

Three Cases
- Case #1: Remove the head of the list
  ```
p = L;
  L = L->link;
  delete p;
  ```
- Case #2: Remove from middle of list
  ```
p = L->link;
  L->link = p->link;
  delete p;
  ```
- Case #3: Remove last element in list
  ```
  Identical to removal of head.
  ```

Ordered Remove

```cpp
bool remove( nodePtr& list, infoType delElem) {
    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.
/* 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)) {
                trail1 = list1;
                list1 = list1 -> link; //while
                trail1->link = list2;
            } //while
            else
                if (lessThan(list2->element, list1->element)) {
                    trail2 = list2;
                    list2 = list2 -> link; //while
                    trail2->link = list1;
                } //else
        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?

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

```cpp
void destroyList( nodePtr& list)
{  nodePtr p = list;
    while (list != NULL) {
        list = list->link;
        delete p;
        p = list;
    }
}
```

```cpp
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
}
```

Linear Linked-List Variations

- **Circular List**
- **Double Linked-List (non-circular)**
- **Circular Double Linked-List**
Linked-List Variation

Declarations

- Double Linked-List Declaration (non-circular)
  ```c
  typedef struct node *nodePtr;
  typedef struct {
    infoType element;
    nodePtr prev, next;
  } node;
  dblList dlist;
  ```

- Double Linked-List Declaration (circular)
  ```c
  typedef struct node *dblList;
  typedef struct {
    infoType element;
    dblList prev, next;
  } node;
  dlist dlist;
  ```

Circular Double Linked-List

Insertion

- Insert 83 into the ordered list:
  ```c
  typedef struct node *dblList;
  typedef struct {
    infoType element;
    dblList prev, next;
  } node;
  dlist 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

Delete 75 from the list:

```
INSERT CODE HERE
```

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)

```
/* 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) */
dblList mergelists(dblList list1, dblList list2)
{
    dblList 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)) {
            trail1 = list1->next; //move list1 up 1 node
            list1->next = list2;
        } else {
            trail2 = list2->next; list2->prev = trail1;
        } //if
    } //while
    last = (list1 == NULL) ? list2 : list1; //nonempty list?
    while (last->next != NULL)
        last = last->next;
    last->next = head;
    head->prev = last;
    list1 = list2 = NULL;
    return head;
}
```