Linked Lists

10.1 Links and Pointers

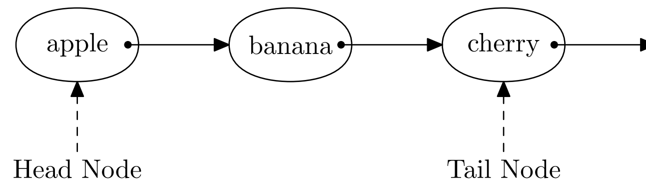
The linked list is not an ADT in its own right; rather it is a way of implementing many data structures. It is designed to *replace* an array.

A linked list is

a sequence of nodes each with a link to the next node.

These links are also called pointers. Both metaphors work. They are links because they go from one node to the next, and because if the link is broken the rest of the list is lost. They are called pointers because this link is (usually) one-directional—and, of course, they are pointers in C/C++.

The first node is called the ***head node***. The last node is called the ***tail node***. The first node has to be pointed to by some external holder; often the tail node is too.



One can use a `struct` or `class` to create a node. We use a `struct`, the predecessor to objects; the syntax is similar to classes (except that its members are public).

```
struct Node {
  <data>
  Node *link;
};
```

(where <`data`> means any type of data, or multiple types). The class using or creating the linked list then has the declaration:

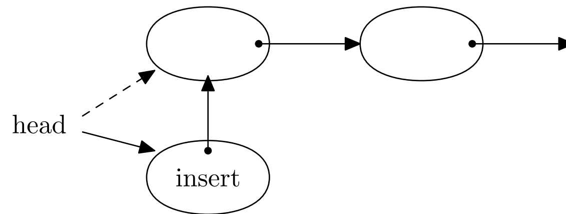
```
Node *head;
```

10.2 Insertion and Traversal

For traversing a list, the idea is to initialize a pointer to the first node (pointed to by `head`). Then repeatedly advance to the next node. `NULL` indicates you've reached the end. Such a pointer/reference is called a ***cursor***. There is a standard construct for a *for-loop* to traverse the linked list:

```
for( cursor=head; cursor!=NULL; cursor=cursor->link ){
  <do something with object referenced by cursor>
}
```

For insertion, there are two separate cases to consider: (i) addition at the root, and (ii) addition elsewhere. For addition at the root, one creates a new node, changes its pointer to where head currently points, and then gets head to point to it.

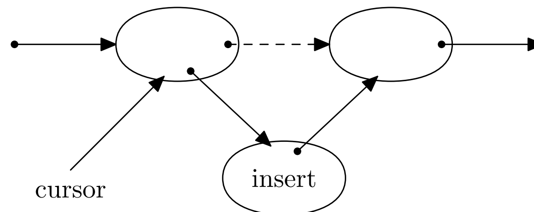


In code:

```
Node *insertPtr = new Node;
update insertPtr's data
insertPtr->link = head;
head = insertPtr;
```

This code also works if the list is empty.

To insert elsewhere, one need a reference to the node **before** where one wants to insert. One creates a new node, changes its pointer to where the node before currently points, and then gets the node before to point to it.



In code, assuming `cursor` references node before:

```
Node *insertPtr = new Node;
update insertPtr's data
insertPtr->link = cursor->link;
cursor->link = insertPtr;
```

10.3 Traps for Linked Lists

1. You **must** think of and test the exceptional cases: The empty list, the beginning of the list, the end of the list.

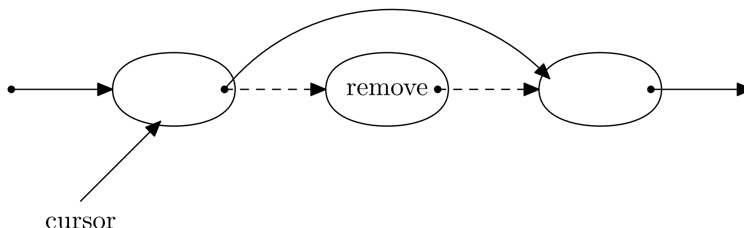
2. Draw a diagram: you have to get the picture right, and you have to get the order right.

10.4 Removal

The easiest case is removal of the first node. For this, one simply advances the head to point to the next node. However, this means the first node is no longer referenced; so one has to release that memory:

```
Node *removePtr = head;
head = head->link;
delete removePtr;
```

In general, to remove a node that is elsewhere in the list, one needs a reference to the node **before** the node one wants to remove. Then, to skip that node, one needs only to update the link of the node before: that is, get it to point to the node after the one wants to delete.



If the node before is referenced by `cursor`, then `cursor->link` refers to the node to be deleted, and `cursor->link->link` refers to the node after. Hence the code is:

```
Node *removePtr = cursor->link;
cursor->link = cursor->link->link;
delete removePtr;
```

The problem is to organize `cursor` to be in the correct place. In theory, one would like to traverse the list, find the node to be deleted, and then back up one: but that's not possible. Instead, one has to look one node ahead. And then beware `NULL` pointers. See sample code.

10.5 And Beyond

Arrays are better at ***random access***: they can provide an element, given a position, in constant time. Linked lists are better at additions/removals at the cursor: done in constant time. Resizing arrays can be inefficient (but is "on average" constant time).

Doubly-linked lists have pointer both forward and backward. These are useful if one needs to traverse the list in both directions, or to add/remove at both ends.

Dummy header/tail nodes are sometimes used. These allow some of the special cases (e.g. empty list) to be treated the same as a typical case. While searching takes a bit more care, both removal and addition are simplified.

Exercise. Develop code for making a copy of a list.

Sample Code: MyLinkedBag

```
// MyLinkedBag.h - wdg 2008
#ifndef MY_LINKED_BAG_H
#define MY_LINKED_BAG_H
#include <iostream>
using namespace std;

typedef int ListType;   // produces a bag of ints
struct BagNode
{
  ListType data;
  BagNode *next;
};

class MyLinkedBag
{
  public:
      MyLinkedBag();
      ~MyLinkedBag();  // should be virtual; discussed later
      void add(ListType x);
      bool remove(ListType x);
      bool isEmpty() const;

      friend ostream & operator<< ( ostream & , const MyLinkedBag & );

  private:
      BagNode *head;      // head of the list
};
#endif
```

```
#include "MyLinkedBag.h"
//#include <cstddef>
//using namespace std;

  // Constructor
  MyLinkedBag::MyLinkedBag() : head(NULL)
  {
```

```
}

// Destructor
MyLinkedBag::~MyLinkedBag()
{
    while( head!=NULL ) {
        BagNode *hold = head->next;
        delete head;
        head = hold;
    }
}

// Mutator methods:
void MyLinkedBag::add(ListType x)   // adds at front
{
    BagNode *newPtr = new BagNode;
    newPtr->data = x;
    newPtr->next = head;
    head = newPtr;
}

bool MyLinkedBag::remove(ListType x)
{
    if( head==NULL )
        return false;
    if( head->data==x ) {
        BagNode *hold = head;
        delete head;
        head = hold;
        return true;
    }
    // so list not empty and not first node to be deleted
    BagNode *cursor = head;
    while( cursor->next != NULL  &&  cursor->next->data != x )
        cursor = cursor->next;
    if(cursor->next == NULL)
        return false;
    else {
        BagNode *hold = cursor->next->next;
```

```
        delete ( cursor->next );
        cursor->next = hold;
        return true;
    }
}

// Accessor methods:
bool MyLinkedBag::isEmpty() const   // Returns: whether the list is empty.
{
    return (head==NULL);
}

ostream & operator<< ( ostream & out, const MyLinkedBag & other)
{
    for(  BagNode *cursor = other.head; cursor; cursor=cursor->next)
        out << cursor->data << " ";
    out << endl;
}
```