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

\[
\text{a sequence of nodes each with a link to the next node.}
\]

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.

\[
\begin{align*}
\text{apple} & \quad \text{banana} & \quad \text{cherry} \\
& \quad \text{Head Node} & \quad \text{Tail Node}
\end{align*}
\]

One can use a struct or class to create a node. We use here a struct. Note that in C++ a struct is identical to a class except that its members are public by default.

\[
\begin{verbatim}
struct Node {
    <data>
    Node *link;
};
\end{verbatim}
\]

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

\[
\begin{verbatim}
Node *head;
\end{verbatim}
\]

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. nullptr 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!=nullptr; 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.

![Diagram of insertion at the root](image)

In code:

```cpp
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.

![Diagram of insertion elsewhere](image)

In code, assuming `cursor` references node before:

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

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

```c
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:

```c
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 `nullptr` 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.

One can also have **circularly linked lists** where the last node points to the first.

Sample Code

```
MyLinkedBag.h
MyLinkedBag.cpp
```