Linked List Variations

1 Introduction

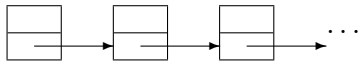
In this handout, we discuss variations on linked lists:

- Doubly-linked lists
- Cyclic lists
- Dummy nodes
- Endogenous lists

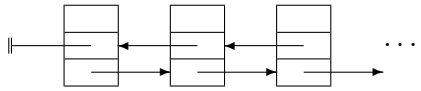
The textbook also does so (section 4.6 on pages 232–234), and we recommend you read that too. All these variations (and combinations thereof) are used in professional practice.

2 Doubly-Linked Lists

A traditional (singly-linked) list has just one link per node:



But this makes it infeasible to transverse the list backwards, such as one may need to do to remove a node. So *doubly-linked* nodes include *two* links per node:



A typical node declaration is as follows:

```
private class Node {
  int data;
  Node prev;
  Node next;
  public Node(int d, Node p, Node n) {
    data = d;
    prev = p;
    next = n;
  }
}
```

When inserting or removing nodes in a doubly-linked list, there are twice as many links to change as with a singly-linked list. If your code is ever changing the "next" field of some node, it should probably also be changing the "prev" field of a related node. This property will help you find bugs in your code. For example, assuming one is inserting a new node at the head of a link, you might write code as follows:

```
head = new Node(xxx,null,head);
// something is missing!
```

Here, we created a link from this node to the former "head" node (assuming it is not null), but the "prev" link of this new node is always null because it is first. However, we are not done and we can tell that we are not done, because we never set the "prev" link of any existing node. Indeed the former "head" node, if it exists, needs to point back to this new node:

```
if (head.next != null) {
  head.next.prev = head;
}
```

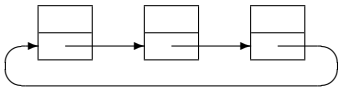
The "prev" and "next" links should be opposites, so that x`.next.prev` $= x$`.prev.next` $= x$ unless the first link is null. The above assignment can be need as effecting this equality.

Self-Test Exercises

1. Is `head.prev.next == head` true after the situation just described?
2. The code given above for inserting at the head of a list doesn't work if there is a "tail" pointer too. Fix it!
3. How can we append a node to the *end* of a list if we have both head and tail pointers? Write the code!
4. Write the code to remove a (non-null) node `p` from a doubly-linked list with head and tail pointers. Make sure to handle all four cases: that `p` is the first node, the last, both or neither.

3 Cyclic Lists

A *cyclic* list is one in which the last node points back to the first one:



WIth a cyclic list, we don't use `null` to terminate a list. We also don't need a `head` field; we can use the `tail` field to find the head node.

Counting the nodes (or any other task that involves looking at all the nodes) in a cyclic list is somewhat tricky:

```
int count = 0;
if (tail != null) {
  ++count;
  for (Node p = tail.next; p != tail; p = p.next) {
    ++count;
  }
}
```

We start at the head (`tail.next`) but as this requires a dereference, we first must check whether the tail is null. If it isn't, then we have to count this node, since the loop will stop when we reach the tail node.

Inserting a node at the head of such a list is almost the same as inserting at the tail. The only difference is that in the latter situation, we adjust the "tail" pointer to refer to the new node.

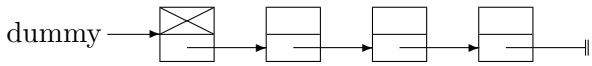
Using cyclic lists reduces the number of cases with null pointers, which makes for less complex programming for insertions and deletions. However, there is still the case of the empty list: inserting something into an empty list, or removing the last node from a list both involve special cases.

Self-Test Exercises

5. Verify the code given for computing the count for the given list with three nodes. Go through the code by hand and check that it indeed only looks at two nodes.
6. Write code to insert a new node at the end of a non-empty cyclic list
7. Add the case that the list might be empty.
8. Write code to remove the first node from a non-empty cyclic list. Make sure to handle the code that the result might be empty.
9. Why is it difficult to remove the *last* node from a singly-linked cyclic list?

4 Dummy Nodes

Another way to reduce/remove special cases from linked lists is to use a *dummy* node, which is a node that does not hold any value, but is in the list to provide an extra node at the front (and/or rear) of the list. So for example, a list with three "real" nodes with an extra dummy node at the front would appear as follows:

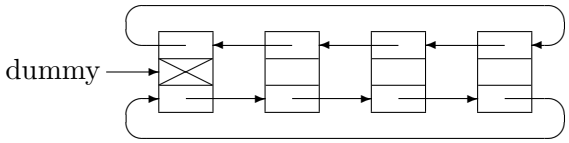


The dummy node is very useful as a "lag" pointer when inserting or removing nodes. For instance, if one wants to remove all the nodes whose data is 0, one can write:

```
Node lag = head;
for (Node p = lag.next; p != null; p = p.next) {
  if (p.data == 0) {
    lag.next = p.next;
  } else {
    lag = p;
  }
}
```

Without a dummy node, this code is much more complex.

It is also possible to put a dummy at the end as well, but more commonly, one uses a dummy node with a cyclic list or even a cyclic doubly-linked list:



The head of the list can be accessed using `dummy.next` and the tail by `dummy.prev`.

When one has a cyclic list with a dummy node (either singly or doubly linked) there are never any null pointers in the data structure. This cuts down the number of cases needed.

Self-Test Exercises

10. Dummy nodes waste space. Explain why this need not concern us.

11. What does the empty singly-linked cyclic list with a dummy node look like? doubly-linked?

12. In what way does a acyclic (list *without* a cycle) still have null pointers even if it has dummy nodes at the front and rear of the list?

13. Suppose we have a singly-linked acyclic list. Having a dummy node at the front reduces special cases (as shown). Why does it not make sense to have a dummy node at the tail, as well?

14. To insert a node after a given node `p` in a doubly-linked cyclic list with a dummy node requires only one case. Write the code!

5 Endogenous Linked Lists

All the linked lists described so far and in the textbook are what we call *exogenous* linked lists, in which the links are stored outside of, that is separate from, the data. We have a separate node structure used internal to the linked structure.

However in system programming, one frequently encounters the other kind of linked list, the *endogenous* linked list, in which the links are stored within the data. In other words, the data being linked already have the ability to be linked together; they already have (say) "prev" and "next" fields. The advantage of such a system is that it does not require one to allocate new objects (the nodes) in order to connect existing objects together in lists. Necessarily each piece can only be placed in one list.

In the kernel of an operating system (such as Windows or Linux), there will be task or process or thread objects. Each object can be "runnable" (in a run qeueue) or waiting in a queue for some event or some resource. Removing an object from one queue and inserting into another doesn't require any object allocation or deallocation, which is good since kernel code can rarely safely do allocation or deallocation. Endogenous lists are also a good match for physical objects that (of course) can only be in one container at a time.

Working with endogenous lists is similar to working with exogenous (traditional) lists except that methods inserting or removing nodes take the nodes in or return them. For example, adding a new node at the head is simple:

```
data.next = head;
head = data;
```

Endogenous lists can be singly or doubly linked, and can be cyclic or acyclic. They can have dummy nodes as well, but in this case, the dummy nodes (often called *sentinels*) often must have valid data fields so that they can function appropriately.

Self-Test Exercises

15. Look up in a dictionary what "endo-" and "exo-" mean.

16. What happens when a node is inserted into one endogenous list and then into another without first removing it from the first? Choose a particular configuration (single/double, cyclic/acyclic, dummy/no dummy) and see what happens.

17. Write code to insert a "task" at the end of an endogenous singly-linked acyclic list without dummy nodes, but with head and tail pointers.