COMP301
Data Structures & Algorithms [C++]

Dr. N. B. Gyan

Central University, Miotso. Ghana

Introduction to Linked Lists

Non-contiguous Lists: Linked Lists

- The idea of non-contiguous lists is one of the techniques often discussed on data structures in computer science.
- Earlier it was shown that, by using the dynamically expanding array, we can read in an arbitrary number of input items. This technique has one serious problem.
- (Have you thought about it yet? If not take a moment to think about it. 😃)

The Problem

Suppose that we are reading 1000-byte records and we have 1,000,000 bytes of memory available. Also suppose that, at some point, the array holds 400 records and is full. Then to double, we create an array of 800 records, copy over 400 records, and then delete the 400 records. The problem is that, in this intermediate step, we have both a 400- and an 800-record array in use and that the total of 1200 records exceeds our memory limit. In fact, we can run out of memory after using only roughly one third of the available memory.

The Solution

- A solution to this problem is to allow the list of records to be stored non-contiguously.
- For each record we maintain a structure that stores the record and a pointer, next, to the next structure in the list.
- The last structure has a `NULL` next pointer. We keep a pointer to both the first and last structures in the list.

The Solution

- The resulting structure is the classic linked list; which stores data with a cost of one pointer per item.

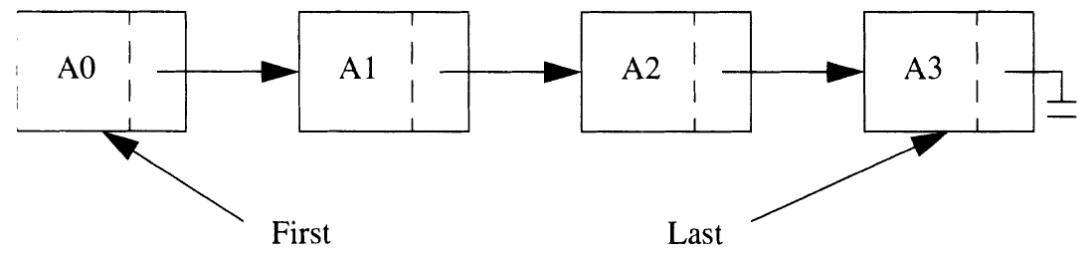


Fig. 2: Illustration of a simple linked list.

Defining the linked-list structure

The structure definition is

```
struct Node
{
  Object item;     // Some element
  Node *next;
};
```

At any point we can print the list by using the iteration

```
for( Node *p=first; p!=NULL; p=p->next )
 printItem( p->item );
```


Defining the linked-list structure

At any point a new last item, **x**, can be added as in

```
last->next = new Node; // Attach a new Node
last = last->next; // Adjust last
last->item = x; // Place x in the new node
last->next = NULL; // It's the last, so make
                   // next NULL
```

- An arbitrary item can no longer be found in one access. Instead, we must scan down the list.

- This difference is similar to that of accessing an item on a compact disk (one access) or a tape (sequential).
- On the other hand, inserting a new element between two existing elements requires much less data movement in a linked list than in an array.
- More on this later...

Abstract Data Types (ADTs)

Abstract Data Types

- An abstract data type (ADT) is a set of objects together with a set of operations. Abstract data types are mathematical abstractions.
- Nowhere in an ADT's definition is there any mention of *how* the set of operations is implemented.
- Objects such as lists, sets, and graphs, along with their operations, can be viewed as ADTs, just as integers, reals, and booleans are data types.

- Integers, reals, and booleans have operations associated with them, and so do ADTs. For the set ADT, we might have such operations as *add*, *remove*, *size*, and *contains*.
- Alternatively, we might only want the two operations *union* and *find*, which would define a different ADT on the set.

The List ADT

Abstract Data Types

- For our purpose we will be dealing with a general list of the form $A_0, A_1, A_2, \ldots, A_{N-1}$.
- This simply means that A_i follows (or succeeds) A_{i-1} $(i < N)$ and that A_{i-1} precedes A_i $(i > 0)$.
- And we say that the size of this list is N. There is also the special list of size 0 which is called an **empty list**.
- The first element of the list is A_0, and the last element is A_{N-1}.
- The **position** of element A_i in a list is i.
- We also assume (for our discussion purposes) that the elements in the list are integers, but in general, any complex elements are allowed.

The List ADT

- Associated with these "definitions" is a set of operations that we would like to perform on the List ADT.
- Some popular operations are
  - `printList` and `makeEmpty`, which do the obvious things;
  - `find`, which returns the position of the first occurrence of an item;
  - `insert` and `remove`, which generally insert and remove some element from some position in the list;
  - `findKth`, which returns the element in some position (specified as an argument).

The List ADT

- If the list is $34, 12, 52, 16, 12$, then `find(52)` might return 2; `insert(x,2)` might make the list into 34, 12, x, 52, 16, 12 (if we insert into the position given); and `remove(52)` might turn that list into 34, 12, x, 16, 12.

Array Implementation of Lists

- All these instructions can be implemented just by using an array.
- Although arrays are created with a fixed capacity, the `vector` class, which internally stores an array, allows the array to grow by doubling its capacity when needed.
- An array implementation allows `printList` to be carried out in linear time, and the `findKth` operation takes constant time, which is as good as can be expected.

Array Implementation of Lists

- However, insertion and deletion are potentially expensive, *depending* on where the insertions and deletions occur.
- In the worst case, inserting into position 0 (at the front of the list) requires pushing the entire array down one spot to make room, and deleting the first element requires shifting all the elements in the list up one spot, so the worst case for these operations is $O(N)$.

Array Implementation of Lists

- On average, half of the list needs to be moved for either operation, so *linear time* is still required. On the other hand, if all the operations occur at the high end of the list, then no elements need to be shifted, and then adding and deleting take $O(1)$ time.
- There are many situations where the list is built up by insertions at the high end, and then only array accesses (i.e., `findKth` operations) occur. In such a case, the array is a suitable implementation.

- However, if insertions and deletions occur throughout the list and, in particular, at the front of the list, then the array is *not* a good option.
- The next section deals with the alternative: the *linked list*.

Linked Lists

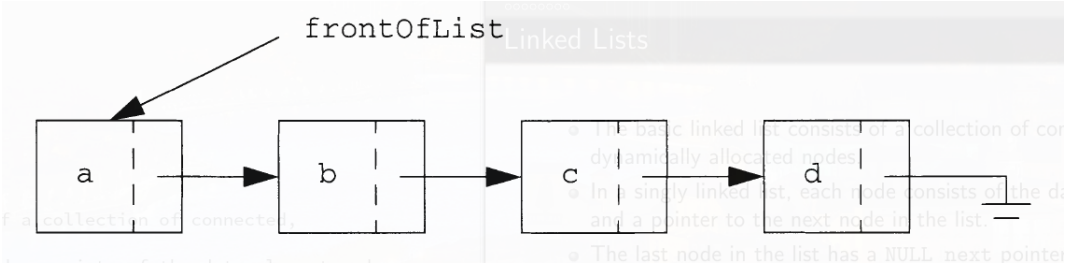
Linked Lists

- The basic linked list consists of a collection of connected, dynamically allocated nodes.
- In a **singly linked list**, each node consists of the data element and a pointer to the next node in the list.
- The last node in the list has a `NULL next` pointer. In this section we assume that the node is given by the following type declaration:

```
struct Node
{
  Object element;
  Node *next;
};
```


Linked Lists

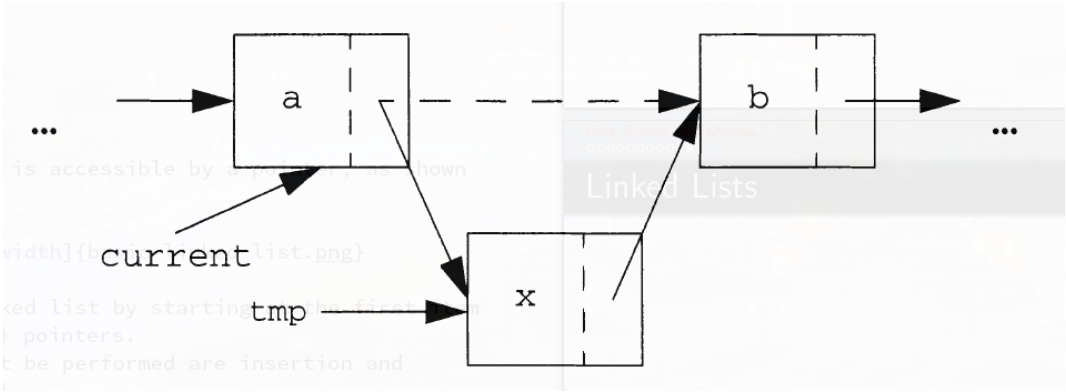
- The first node in the linked list is accessible by a pointer, as shown in the Figure below:



- We can print or search in the linked list by starting at the first item and following the chain of `next` pointers.
- The two basic operations that must be performed are insertion and deletion of an arbitrary item `x`.

Linked Lists

- For insertion we must define where the insertion is to take place.
- If we have a pointer to some node in the list, the easiest place to insert is immediately after that item.
- The Figure below shows how we insert **x** after item a in a linked list.



Linked Lists

- The steps involved are as follows:

```
// Get a new node from the system
tmp = New Node;
// Place x in the element member
tmp->element = x;
// x's next node is b
tmp->next = current->next;
// a's next node is x
current->next = tmp;
```

Linked List Insertion

- As a result of these statements, the old list ...a, b, ...now appears as ...a, x, b, ...
- The code can be simplified if the `Node` has a constructor that initializes the data members directly.
- In that case, we obtain

```
// Get new node
tmp = new Node( x, current->next );
// a's next node is x
current->next = tmp;
```

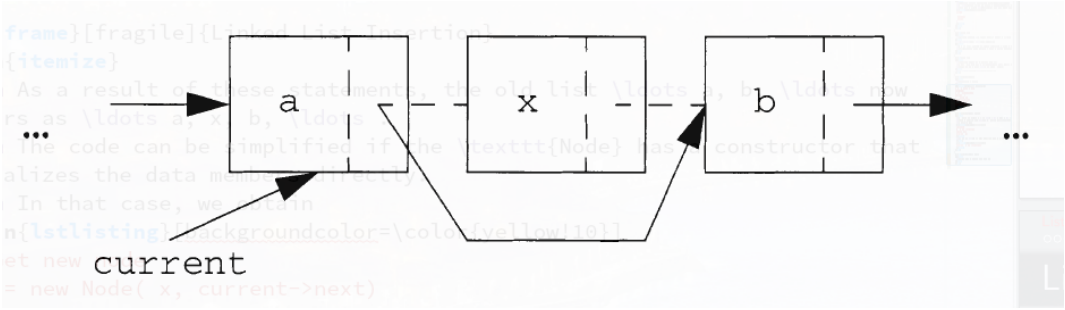

Linked List Insertion

- This shows that `tmp` is no longer necessary. Thus we have the one-liner:

```
current->next = new Node( x, current->next );
```

Linked List Deletion

- The remove command can be executed in one pointer move.



- To remove item `x` from the linked list, we set `current` to be the node prior to `x` and then have `current`'s `next` pointer bypass `x`.

Linked List Deletion

- This operation is expressed by the statement

```
current->next = current->next->next;
```

- The list ...a, x, b, ...now appears as ...a, b, ...

Avoiding Memory Leaks

- A problem with this implementation is that it leaks memory: The node storing `x` is still allocated but is now **unreferenced**.
- By saving a pointer to it first and then calling `delete` after the bypass, we can reclaim the memory:

```
// Save pointer
Node *deleteNode = current->next;
// Bypass the node
current->next = current->next->next;
// Free the memory
delete deletedNode;
```


Avoiding Memory Leaks

The fundamental property of a linked list is that changes to it can be made by using only a constant number of data movements, which is a great improvement over an array implementation.

Using Header Nodes

Using Header Nodes

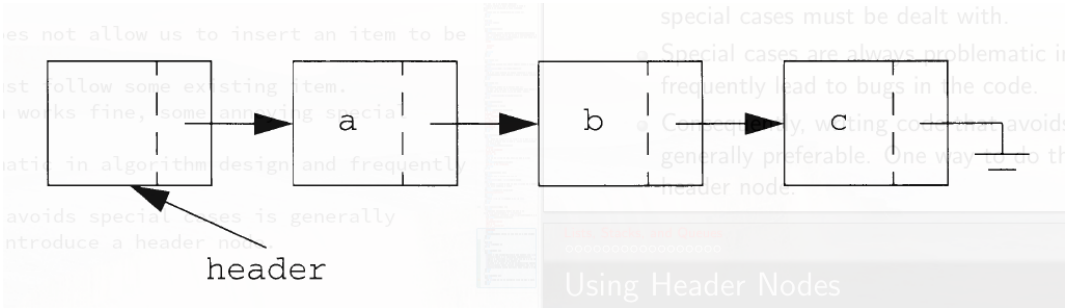
- There is one problem with the basic description: It assumes that whenever an item x is removed, some previous item is always present to allow a bypass.
- Consequently, removal of the first item in the linked list becomes a special case.
- Similarly, the insert routine does not allow us to insert an item to be the new first element in the list.
- The reason is that insertions must follow some existing item.

Using Header Nodes

- So, although the basic algorithm works fine, some annoying special cases must be dealt with.
- Special cases are always problematic in algorithm design and frequently lead to bugs in the code.
- Consequently, writing code that avoids special cases is generally preferable. One way to do that is to introduce a **header node**.

Using Header Nodes

- A **header node** is an extra node in a linked list that holds no data but serves to satisfy the requirement that every node containing an item have a previous node in the list.
- The header node for the list *a*, *b*, *c* is shown below:



Using Header Nodes

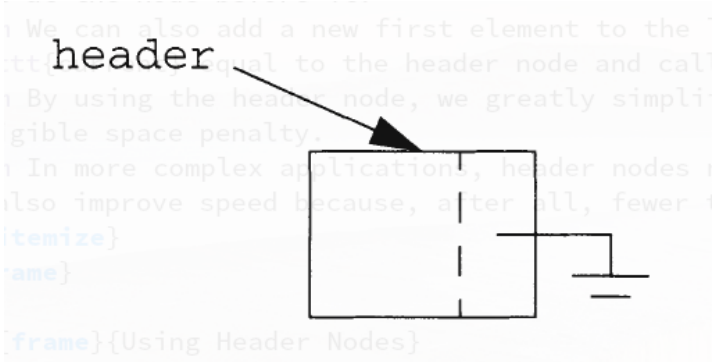
- Note that by this method *a* is no longer a special case.
- It can be deleted just like any other node by having `current` point at the node before it.
- We can also add a new first element to the list by setting `current` equal to the header node and calling the insertion routine.
- By using the header node, we greatly simplify the code—with a negligible space penalty.
- In more complex applications, header nodes not only simplify the code but also improve speed because, after all, fewer tests mean less time.

Using Header Nodes

The following therefore become applicable when header nodes are in use.

- The printing routine must skip over the header node, as must all searching routines.
- Moving to the front now means setting the current position to `header->next`, and so on.
- Furthermore, with a dummy header node, a list is empty if `header->next` is `NULL`.

Using Header Nodes



See you next week, God willing 🙏