# Linked Lists

# Lists

Lists are ordered collections of objects. This does not mean that lists are sorted, but rather that as items are added, one can predict their relative positions. For example, one can define an operator that adds items at the end of the list, or at the front. Given the list 20,30,50, the operation *append 69* will result in 20,30,50,69, while *addfront 12* to the new list would result in 12,20,30,50,69. As you observe this particular list is already sorted. Now to insert element 42 in its proper position in the list, we may like to define a new operation *insert 42*, which would result in the list 12,20,30,42,50,69. If we are searching for a specific item, and we find it in the list, we return 1. The operation *search 30* would return 1, as the element is present in the list. We can also remove items from the list through a delete operation such as *delete 20*, which would create an empty position (a hole) in the list, and all elements after 20 would have to be shifted one position forward to result in the final list 12,30,42,50,69. The effect of *insert 25* would require shifting all elements after 12 one position back to accommodate the new element 25.

You know how easy it is to implement the lists using arrays, where we could do searching or sorting operations. If the list were sorted, the searching operation would take O(log n) operations. What about insert and delete operations? To insert a new element in the beginning of the array, you have to shift all the elements down by one position to accommodate it. Similarly, to delete an item, you have to shift up all the elements to fill in the hole created by the item. For a list with n elements, these operations could be more expensive computations of the order O(n).

Another shortcoming with arrays is that they have to have a fixed size, which must be declared in advance before the program starts executing. In many applications, the number of elements in the list may vary greatly, depending on the input, so memory requirements may change during execution of the program.

# Recursive data structures

What is desired is a way to allocate memory for new items ONLY WHEN necessary, and to free that memory space when it is no longer needed. Furthermore, this newly allocated memory must be logically combined into a single entity (representing the list).
What is a list? We can think of a list as an entity containing a similar entity. A list is a data item followed by another list. The list 12,20,30,42,50,69 can be thought of as being

composed of the data item 12, followed by another list 20,30,42,50,69. Thus we can define a list structure in terms of itself. Data types that are defined in this way are called recursive data types.
Consider the structure

```
struct list{
     int data;
     struct list *next;
};
```

This structure defines a list as containing an item of type int, followed by another list. The second list is connected through the link *next*. More appropriately, this list is referred to as a **linked list**. Each element is linked to the next one through a link field, which is a pointer to the address of the next element.

But the definition alone is not sufficient to store the values of the elements of the list. You have to request memory to allocate space to hold each item. As the structure is dynamic (it can grow or shrink), more memory can be requested dynamically.

# Dynamic Memory Allocation

Creating and maintaining dynamic data structures requires dynamic memory allocation – the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed.

In C, functions `malloc` and `free`, and operator `sizeof` are essential to dynamic memory allocation.

- Unary operator `sizeof` is used to determine the size in bytes of any data type.
  e.g.
  ```
  sizeof(double)
  sizeof(int)
  ```

- Function `malloc` takes as an argument the number of bytes to be allocated and return a pointer of type `void *` to the allocated memory. (A `void *` pointer may be assigned to a variable of any pointer type. ) It is normally used with the `sizeof` operator.