# Data Structures
Spring 2009

Kishore Kothapalli

# Chapter 5

# Linked Lists

## 5.1 A Gentle Introduction

So far we have seen the array as a data structure with different access mechansism: the random access mechanism, the LIFO access mechanism (stack), and the FIFO access mechaninsm (queue). While these are powerful abstractions, one of the drawbacks in the techniques seen so far is that the maximum size of the underlying array has to be known apriori. One may think of a dynamic allocation of an array using programming language construts such as `malloc`, but a size has to be speicified at that point. To clearly see the drawback, consider the following analogy.

Suppose that we have access to a petabyte of storage and we manage a storage company. Our business is to store data of various clients for a small fee. Now let client1 store 100 MB on day one. We give him space from byte 0 to byte 100 MB. Then client2 is given space from 100 MB+1 byte to 150 MB. This is smooth for a while. Now, client 1 wants to extend his 100 MB storage space by adding 2MB to it. Where do we allocate this 2 MB? Since this is an extension the origninal 100 MB, it may be better if we give 102 MB contiguously. However, it is not possible as the contiguous space next to the origninal 100 MB is not free. To relocate the 100 MB or to move other parts and create a space of 2 MB next to the 100 MB of client 1 is also not a practical solution. A clever way out of this is to store that the additional 2 MB starts from byte numbered $y$. This allows the possibility for future extensions also. The only question we have to answer is to see where we should keep note of this fact. A natural place is the last few bytes of the first allocation shall be reserved to hold the location of the next allocation. The address of this next allocation can be treated as a pointer in programming parlance.

To overcome this drawback, we have to think of data structures whose size can grow as required by the application. This percludes any contiguously allocated data structures such as the arrays. So one has to use pointer data structures so that the size used by the data structure can grow dynamically. A linked list is such a data structure. In this chaper, we will study the linked list as a data structure, the operations supported, and applications of linked lists.

## 5.2 The Linked List Data Structure

The linked list is a way of organizing a list of elements with basic operations `insert` and `delete` and other operations such as `find` and `print`. Though an array can be used to support these operations, it can be seen that each operation requires a time $O(n)$, where the array has $n$ elements. Moreover, if we extend the semantics of the `insert` operation to insert not just at the end but also at any point, then such an operation is very impractical.

The linked list is a list of nodes not necessarily contiguously allocated in memory but are linked together so that each node contains information required to access the next node. This information is also called as the variable `next`. So, a node in a linked list looks as shown in Figure **??**.

The linked list data structure supports the following operations. We assume that the location of the first node is given by the special vairable called `head`.

- `Insert(x, a)` : Insert $x$ next to element $a$ in the list.
- `Remove(x)`: Remove element $x$ from the list.
- `Find(x)` : Find and return, if present, the location of element $x$.
- `Print()` : Print the contents of the list.

Using C–style pointers, the following is the pseudocode for the operations. We assume that we have a strcuture such as:

```
struct node {
    int value;
    struct node *next;
}
```

With the above definition of a structure, let us implement the above operations. Given a linked list via the head pointer, let us first implement the operations `Find` and `Print`.

```
Algorithm Find($x$)
begin
tempHead = head;
while (temphead != NULL) do
    if temphead ->data == $x$ then
        return temphead;
    temphead = temphead ->next; end-while
return NULL;
end
```

```
Algorithm Print()
begin
temphead = head;
while (temphead != NULL)
    Print(temphead ->data);
    tempead = temphead ->next;
end-while
end
```

The insertion and removal of items from a linked list requires that we have a pointer to the place of insertion/deletion.

## 5.3 Application 1 – Stacks and Queues

We noticed in the previous week that a stack and a queue can be implemented using an array. However, the fact that one has to specify upfront the maximum number of elements that the stack can contain, is a limitation. To overcome that problem, one can use a linked list to implement a stack. The head of the list can be thought of pointing to the top of the stack. The Push operation adds an element to the list at the head. The Pop operation deletes the element at the head of the list.

For a queue however, we need both the front and the rear. For this purpose, one can think of maintaining both the head and the tail of the list. The front of the queue can be the tail and the rear of the queue can be the head of the list.

## 5.4 Application 2 – Polynomials

We now use linked lists to perform operations on polynomials. Let $f(x) = \sum_{i=0}^{d} a_i x^i$. The quantity $d$ is called as the degree of the polynomial, with the assumption that $a_d \neq 0$. A polynomial of degree $d$ may however have missing terms – i.e., powers $j$ such that $0 \leq j < d$ and $a_j = 0$.

The standard operations on a polynomial are addition and multiplication. If we store the coefficient of each term of the polynomials in an array of size $d + 1$, then these operations can be supported in a straight-forward way. However, for sparse polynomails, i.e., polynomials where there are few non-zero coefficients, this is not efficient.

One possible solution is to use linked lists to store degree, coefficient pairs for non-zero coefficients. With this representation, it makes it easier if we keep the list of such pairs in decreasing order of degrees.

## 5.5 Application 2 – Sparse Matrices

## Acknowledgements

Prof. Viswanath shared the story of storage company during a discussion.

## 5.6 Tutorial, Lab, and Homework

- Discuss tail recursion
- implement linked lists, doubly linked list (in tutorial)
- Implement stack and queue using linked lists.
- Timing of recursive find using tail recursion and compare wrt non-recursive version.
- HW : implement polynomial and operations (add, multiply, power)
- HW : implement sparse matrices : 2 variations. comp-row and block diagonal.
- HW: Additional operations on linked lists: merge, remove duplicates, intersection, ...
- HW :