Lecture 12
Doubly Linked Lists
(with Recursion)

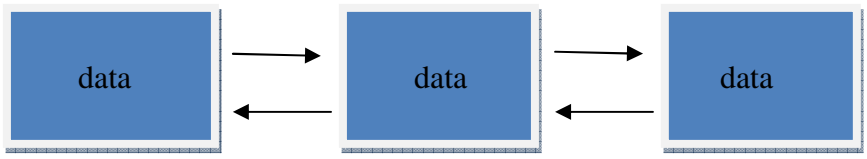
In this lecture

- Introduction to Doubly linked lists
- What is recursion?
- Designing a node of a DLL
- Recursion and Linked Lists
    - Finding a node in a LL (recursively)
    - Printing a LL (recursively)
    - Appending a node to a LL (recursively)
    - Reversing a LL (recursively)
- Further Readings
- Exercises

Introduction to Doubly Linked Lists

As we learnt before, singly linked list is a structure that contains at least two fields, data field and a pointer to the next node. Singly linked lists are flexible structures where memory can be allocated in "small" blocks as needed. Also, when deleting or inserting nodes from a singly linked list, the overhead is relatively low compared to array insertions where array elements have to be moved with a greater cost. This makes singly linked lists (LL) preferred data structures to use in many applications.
However, one drawback in LL is that list can only be navigated in one direction and all navigations must start from the head node. But imagine an application where nodes needs to be inserted quickly, in order, and the data file typically provides clusters of data that needs to be inserted close to each other. It is quite common in data sets to have related data close to each other. For example, think of a user database where names are almost sorted and therefore needs to be inserted next to each other. In this case, it makes sense for the current pointer to remain in place until the next value is read from the data set. Based on the value of the date set, the current pointer can move forward or backwards to insert data in correct place.

A **doubly linked list** is a collection of objects linked together by references from one object to another object, both forward and backward. By convention these objects are named as **nodes**. So the basic DLL is a collection of nodes where each node contains one or more data fields AND references to **next node** and **previous node**. The next pointer of the last node points to a NULL reference and the previous pointer of first node points to NULL to indicate the end of the list in both ways.



The entry point into a DLL is always the first or head of the list. It should be noted that head is NOT a separate node, but a reference to the first Node in the list. If the list is empty, then the head has the value NULL. Unlike Arrays, nodes cannot be accessed by an index since memory allocated for each individual node may not be contiguous. We must begin from the head of the list and traverse the list sequentially to access the nodes in the list. Insertions of new nodes and deletion of existing nodes are fairly easy to handle. Recall that array insertions or deletions may require adjustment of the array (overhead), but insertions and deletions in linked lists can be performed very efficiently.

Designing a Doubly Linked List Node

A doubly linked list node contains, minimally a data field and two pointers, one to next node and other to previous node. The following struct can be used to define a DLL node

```
typedef struct DLLNode {
    void* data;
    struct DLLNode *next;
    struct DLLNode* prev;
} DLLNode;
```

Introduction to Recursion

Recursion is the process of an object to refer to another object that may have the same structure. Recall that the definition of node in a singly linked list is called a **recursive data structure**. That is, the definition of the node includes a reference to itself. This allows us to create a linked list, where each node points to another node just like that. Therefore, linked lists are ideal data structures for implementing recursive algorithms.

Recursive algorithms are elegant and allow us to solve complex problems that are otherwise may be difficult to solve. Another benefit of recursion is that the solution to the problem is constructed using a solution to a simple case, and a solution to a recursive problem. Recursion is fairly close to mathematical induction where a theorem is proved by first verifying a simple case, and then showing that the theorem holds for the value n+1, assuming that the theorem holds for some value n, Therefore in recursive algorithms being able to enumerate the problem is critical. Linked lists are ideal structures for recursive algorithms as one can think of nodes can be enumerated from 1 to n (or 0 to n-1).

Recursion and Linked Lists

Let us begin with a simple definition of a recursive search in a LL. Suppose we are looking for a node in a list. Then either the search node is the first one on the list or node must be on a smaller list (that is a list that contains all nodes but the first node, we typically call this the **tail of the list**). We can express this fact more mathematically as follows:

	found	**if target = first of LL**
search(LL, target) =	**not found**	**if LL = NULL**
	search(tail(LL), target)	**otherwise**

Note few things about this definition. The solution to the problem is described simply using what we call a base case. Then the solution to a more general case is described. That is, the solution to searching a list with n nodes is described using solution to the problem of searching a list with n-1 nodes.

Base Case

Any recursive algorithm ***must have a base case***. This is typically the simplest case, such as what is the solution if n = 0 or n =1 or list is empty etc. The base case plays a major role in ending a sequence of recursive calls. Note that in the above example of search we have two outcomes, either the target is found when target is the first node or target is not found, if the list is NULL.

Recursive Case

This is the case where solution to a larger problem is described using a solution to a smaller case. For example, we ask the question, if we know the solution to n-1^{th} problem, can we find the solution to the n^{th} problem (or vice versa). Note in the above case, we state that if first node is not the target, then we can "reduce" the search space by 1, by searching the tail of the LL. This guarantees that we will eventually find the node or we will reach the case, where our search list has only one node or none.

Implementation

Once we have the correct recursive definition, then implementation is straight forward. We simply convert the definition into a function as follows. (We assume that our node is a node that contains string as its data field. We should change the function header as necessary for the given node type)

```
int searchRecurse(node* head, string target) {
   if (head == NULL)  return 1;   /* search fail. Target not found */
   else if (strcmp(head→data,target) == 0)
      return 0; /* search success. Target found */
   else
    return searchRecurse(head→next, target);
}
```

Tracing the Recursive Algorithm
Suppose we need to trace this algorithm to see how this might work. Consider the following list of nodes. For simplicity we have only shown names and links.

andy → guna → hill → ian → zoo → NULL

Suppose we are looking for the target "zoo". Here is how we trace the recursive code. Note that the **bold faced** first argument represent the whole LL.
searchRecurse(**andy**, "zoo") → searchRecurse(**guna**, "zoo")
→ searchRecurse(**hill**, "zoo") → searchRecurse(**ian**, "zoo")
→ searchRecurse(**zoo**, "zoo") → 0 (success)
We note that the final call searchRecurse(**zoo**, "zoo") result in a success since first node of the list **zoo** is the same as the target "zoo".

Exercise: Draw the trace of the call **searchRecurse**(**andy**, "billy"). In this case you must return 1 for failure.

Cost of Recursion
Although recursion seems elegant, and closer to how we think about solving a problem Mathematically, there is a significant cost to recursion. The cost of recursion is associated with the many calls that are made using the run time stack. As functions calls other functions, the run time stack tends to grow. If we are making many recursive calls, it is possible that recursion may affect the working memory and therefore may slow down the program or crash. Hence the decision to use of recursion must be carefully weigh against the available resources. If the available resources are minimal, you must avoid the recursion at all cost.

Other Examples of Recursion
Given below are more examples of recursion for some standard LL operations. We note that all these definitions apply to any type of LL, including DLL's.

Example 2
Definition

$$InsertToEnd(LL, target) = \begin{cases} head = target & If\ LL = NULL \\ insertToEnd(tail(LL), target) & otherwise \end{cases}$$

Implementation
```
node insertToEnd(node* head, node* target) {
   if (head == NULL) {
      head = target;
      return head;
   }
   return insertToEnd(head→next, target);
}
```

Tracing
insertToEnd (**andy**, "billy") → insertToEnd (**guna**, "billy")
→ insertToEnd (**hill**, "billy") → insertToEnd (**ian**, "billy")
→ insertToEnd (**zoo**, "billy")

Example 3

Reversing a LL using recursion requires a definition as follows. If the list is empty or has only one node, then return the same list (this is the base case). If the list contains more than one node, then we can define reverse as follows.

$$\text{reverse}(LL) = \begin{cases} LL & \text{if LL = empty} \\ \text{insertToEnd}(\text{reverse}(\text{tail}(LL)), \text{firstof}(LL)) & \text{otherwise} \end{cases}$$

Implementation

```
node* reverse(node* head) {
   if (head == NULL) {
      return head;
   }
   else {
     return insertToEnd(reverse(head➔next), head))
    }
}
```

Example 4

The following code displays printing a LL recursively.

```
int printLL(node* head){
   if (head == NULL)
       printf("NULL\n");
   else {
     printf("%d ➔", *(head➔data));
     printLL(head➔next);

   }
}
```

- Further Readings
- Exercises