Computer Science 1MD3

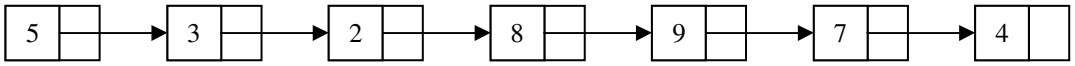
Lab 6 – Dynamic Lists/Linked Lists (Finally)

Up until now all the lists that we worked with were typically stored in arrays. However arrays are not very good at dynamically storing data. In other words, procedures required to add and delete elements in an array are very inefficient. If we instead use a linked list we find that these same procedures become a lot faster. This lab will primarily focus on the underlying theory of linked lists and their implementation in C.

WHAT IS A LINKED LIST

Imagine a structure that has two components: a value, and a pointer. If every value in the list pointed to the value that came after it, this would constitute a list.

Linked list 1.1



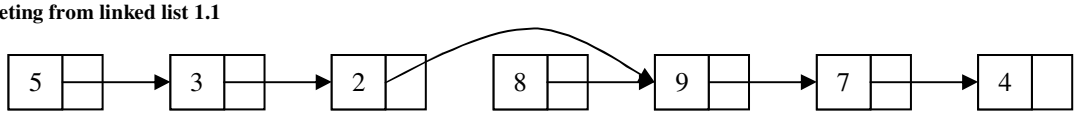
This is how you could visualize a linked list.

DELETING

Deleting a list value from an array would involve removing the element, bumping every other element down one and then modifying the length of the array.

In a linked list, since we are using pointers, we can simply "point" around the element that we are trying to delete.

Deleting from linked list 1.1



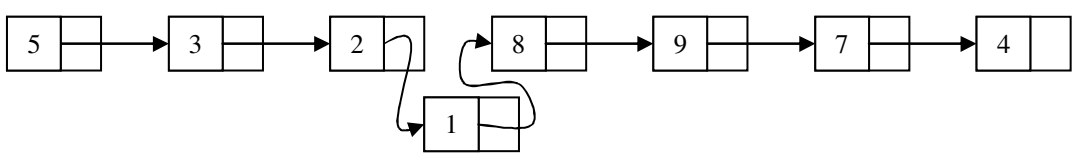
This is an example of deleting 8 from the linked list. Please note that 8 maintains its pointer to 9, however since nothing is pointing to 8, 8 is considered out of the list. We will learn to give the memory allocated for storing 8 back to the system, completely deleting it.

INSERTING

To insert in an array you would have to insert the element, bump the following elements up by one then increase the array size by one.

In a linked list you simply modify pointers to include the new element.

Inserting into linked list 1.1



This is an example of inserting 2 at position 4. Note that 2's pointer was modified to point to 1, and 1 was set to point to 8.

SEARCHING

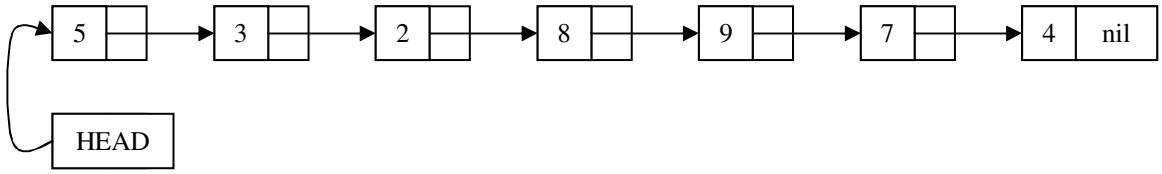
Searching is about the only thing that an array can do more efficiently. Say we would like to access array element at 8, we could simply do `A[8]`.

In a linked list, we have to start at the beginning of the list and advance 8 times returning the value we stop at.

IMPLEMENTING

There are two things that must be dealt with when implementing a linked list in C, where to start, and where to end. We start at what is called the *head pointer*, which is a pointer, `head`, pointing to the first value in the list. To end the list we will adopt the convention of having the last element in the list point to `nil`.

Implementation



It will also be necessary to create a structure like we mentioned, which will store a value and a pointer to the next list item. This structure typically looks like:

List structure

```
/1/    struct list {
/2/          data_type val;
/3/          struct list *next;
/4/    };
```

The `date_type` in line 1 should correspond with the data that you desire to store. We will assume an integer list with the following structure to illustrate some examples.

Integer list structure

```
/1/    struct list {
/2/          int val;
/3/          struct list *next;
/4/    };

/5/    typedef struct list listType;
```

Recall that when we are working with pointers to structures we can refer to the data in the structure like:

```
ListType *x;
x->val;
x->next;
```

We will now design some procedures, which will be important when dealing with linked lists. These procedures are `new`, `insert`, `delete`, and `makenull`, which respectively create a new list, insert an element into a list at a given position, delete an element at a given position, and nullifies the list.

MAKENULL

The `makenull` command will take the head pointer to the list and make it point to nothing, and store nothing. This analogous to the `emptylist` procedures used for arrays.

MakeNull

```
void makeNull (listType *x) {

      x->val=NULL;
      x->next=NULL;

};
```

NEW

The `new` function will accept an integer value and reserve the necessary memory for storing the value in the list structure, returning a pointer to where it was stored.

New

```
listType* new (int x) {

    listType *L;

    L = (listType*)malloc(sizeof(listType));
    L->val=x;
    L->next=NULL;

    return L;

};
```

Investigating this procedure we see that we first use `malloc` to reserve some memory for the structure, then we put the desired value in it, and then by default we make it point to nothing. The return statement returns the pointer to this structure.

INSERT

The `insert` procedure will insert integer `a` at position `p`, in linked list `x`.

insert

```
    void insert (int a, int p, listType *x) {

        listType *temp;
        int count;

        for (count=1; count<p; count++) {

            x=x->next;
            if(x->next==NULL) break;

        }

        temp=x->next;
        x->next=new(a);
        (x->next)->next=temp;

    };
```

The main idea of this procedure is to start at the header and move to the next element `p-1` times. The `p-1` element is then set to point to the new value, which ends up at position `p`, this new value is then set to point to the value that `p-1` used to be pointing to (refer back to the visual representation of insert). Notice that if `p` exceeds the actually length of the list, the new element is just put at the end of the list.

DELETE

The `delete` function will delete element `p` from list `x`.

Delete

```
    void delete (int p, listType *x) {

        int count=1;
        listType *temp;

        for (count=1; count<p; count++) {

            if (x->next==NULL) return;
            x=x->next;

        }

        temp = x->next;
        x->next=(x->next)->next;
        free(temp);

        return;

    };
```

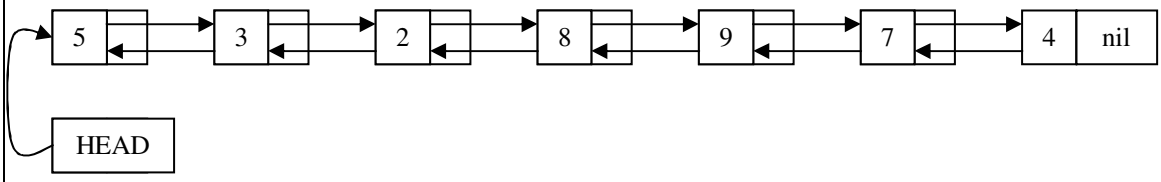
Much like in insert we advance through the list till element `p-1`, we then make `p-1` point to what `p` is pointing to, which deletes `p` from the list (refer back to the visual example). We also use the stdlib.h function 'free()' to return the allocated storage of the deleted node to the system. If `p` exceeds the length of the list nothing is deleted.

DOUBLE LINKED LIST

The list structure we have designed limits us to only moving forward in the list, a double linked list is a linked list to the next position and the previous position. This type of list will allow us to move forward and backward which may be useful.

The list would visually look like this:

Double linked list



With a corresponding data structure:

List structure

```
/1/    struct list {
/2/           data_type val;
/3/           struct list *next;
/4/           struct list *back;
/5/    };
```

EXERCISES

Download the file `linkedlist.c` off c-submit.

Using the functions we just defined, design the following functions:

`int end (listType *L);` which returns the length of a given linked list.

`void printList (listType *L);` which prints out a given linked list.

`listType* quickFill (int *A, listType *L);` which takes an array and converts it to link list format.

`void purge (int a, listType *L);` removes all instances of `a` from linked list `L`.

For example, purging 2 from: `2 2 4 5 6 2 1 4 6 8`
gives: `4 5 6 1 4 6 8`