Discussion of linked lists
for csse2310/7231
Joel Fenwick
[The following code has not been tested properly]

One dimensional arrays:

- all elements are stored in a single allocation.
  - Malloc allows size to be decided at run time but you still need to know how big the array needs to be.
  - If the array needs to be a different size, then allocate a new array and copy the contents over from the old array.
- Very fast to access any element in the array [follow the pointer and jump to the offset].
- Insert or removes from anywhere but the end requires shuffling other elements around.

Linked lists:

- Data (often a pointer) is stored in a struct called a node.
- Each node is allocated separately.
- Nodes need to be explicitly tied together [nodes hold pointers to other nodes].
- Individual nodes can be added and removed without moving the others.

```
typedef struct ln {
   void* data;
   struct ln* next;
} Node;

typedef struct {
   Node* head;
   Node* tail;        /* technically only need the head */
} List;
```

With any structure which uses pointers you should write allocate and cleanup functions. [In OO terms constructors and destructors].

```
List* create_list(void) {
   List* n = malloc(sizeof(List));
   n->head=n->tail=0;     /* Always make sure pointers aim at something sensible */
   return n;
}
```

We will come back to destroy a bit later. To add to the end of a list there are two cases: empty list and non-empty list.

```
void append(List* l, void* d) {
   Node* n= malloc(sizeof(Node));
   n->next=0;
   n->data=d;
   if (l->head==0) {    /* list was empty */
      l->head = l->tail=n;
   } else {
      /* n will be the new tail */
      l->tail->next=n;
      l->tail=n;            /* Careful to get these in the correct order */
```

```
    }
}
```

The first node node in the list is l->head, the second node is l->head->next and the third node is l->head->next->next.
So we can find any node in the list by starting at the head and following next pointers. Now we can write our dealloc function.

```
void destroy_list(List* l) {
    /* need to destroy all nodes first */
    /* note that we don't free the data pointer because we might not own it */
    Node* c=l->head;
    while (c!=0) {
        Node* temp=c;    /* be sure you understand why we need temp */
        c=c->next;
        free(temp);
    }
    free(l);
}
```

If you wish to find and destroy nodes:

Note:

1. You need to know the node in the list before the one you will delete.
2. There are special cases (delete head, delete tail, delete only item).

```
void delete(List* l, void* v) {
   if (l->head->data==v) {    /* delete the head of the list */
      if (l->tail->data==v) {  /* only item in the list */
         free(l->tail);
         l->tail=l->head=0;
      } else {
         Node* n=l->head;
         l->head=l->head->next;
         free(n);
      }
   } else {
      Node* t=l->head;
      Node* prev=t;
      if (l->next==0)   {         /* Only one thing in the list and it's not v */
         return;
      }
      t=t->next;
      while (t!=0 && t->data!=v) {
         prev=t;
         t=t->next;
      }
      if (t==0) {                 /* Item is not in the list */
         return;
      }
      prev->next=t->next;
      free(t);
   }
}
```

Work through the above code, draw pictures, make sure you understand it.
Some of the complexity in the above code is caused by the fact that you can only walk the list one way. One way to address this is to add a previous pointer to each node. This arrangement is called a doubly linked list. Note that code which implements this must keep track of the extra pointers.

To bring function pointers into the list:

```
typedef struct {
   int id;
   char* name1;
   char* name2;
   /* other details */
} Student;
```

Suppose we have a list of Student* and we want to search the list for a student with a particular id. The list stores data as void* and we don't want to put Student specific code into the List functions.

```
/* returns true if the student's id matches the parameter */
int comp_student(Student* s, int id) {
   return (s->id==id);
}
```

This function still has Student in its interface though.

```
int comp_student(void* v, void* i) {
   Student* s=(Student*)v;
   int* id=(int*)i;
   return (s->id == *id);
}
```

This function does the same job but has a very generic interface. Now we write a search function for the list.

```
void* search(List* l, int (*f)(void*, void*) , void* d) {
   Node* n=l->head;
   while (n!=0) {
      if (f(n->data, d))
      {
          return n->data;
      }
       n=n->next;
   }
   return 0;
}
```

This function would be called like this:

```
Student* s;
int i=7;
s=search(l, comp_student, &i);
```

Now suppose that we wanted to search for Students with a particular name1.

```
int comp_name(void* v, void* i) {
   Student* s=(Student*)v;
   const char* id=(const char*)i;
   return !(strcmp(s->name, id);
}
```

Using this function to search would look like:

```
Student* s;
s=search(l, comp_name, “Joel”);
```