CS 137 Part 10

Linked List
This Week

- This week we will introduce a common data structure called a linked list.
- It is a structure where the data is in individual pieces chained together it easy to insert new elements.
- We will revisit the idea of storing a polynomial
A linked list consists of

1. A piece of data (I'll use an integer)
2. A pointer to the next Linked List element

```c
struct ll{
    struct llnode *head;
};

struct llnode{
    int item
    struct llnode *next;
};
```
linked list picture

https://www.tutorialspoint.com/data_structures_algorithms/linked_lists_algorithm.htm
Polynomial Picture
Polynomial Struct

/*
Order polynomial so largest degree is at the beginning. Need degree, coefficient, and pointer to next term.
*/

typedef struct polynode {
    int deg;
    double coeff;
    struct polynode *next;
} polynode;
typedef struct poly {
    struct polynode *head
} poly;
Methods

poly *polyCreate();
poly *polyDelete(poly *p);
poly *polySetCoeff(
    poly *p, int deg, double coeff);
double polyEval(poly *p, double x);
int polyDegree(poly *p);
poly *polyReverse(poly *p);
/*
Pre: None
Post: Creates a null polynomial
*/
poly *polyCreate();

/*
Pre: *p is a valid polynomial (even null)
Post: Destroys the polynomial and returns the null polynomial
*/
poly *polyDelete(poly *p);
More

/*
Pre: poly *p is valid
Post: Returns p(x)
*/
double polyEval(poly *p, double x);

/*
Pre: poly *p is valid, deg is nonnegative
Post: Sets the coefficient at degree to be coeff
*/
poly *polySetCoeff(
    poly *p, int deg, double coeff);
More

/*
Pre: poly *p is valid
Post: returns largest nonzero entry in poly
*/
int polyDegree(poly *p);

/*
Pre: poly *p is valid
Post: returns a polynomial copy of it.
*/
poly *polyCopy(poly *p);
Polynomial create and delete are left as exercises.

```c
// Note p is passed *by value*
double polyEval(poly *q, double x) {
    double f = 0.0;
    polynode *p = q->head;
    // iterate over the nodes terms) and evaluate each appropriately
    for (; p; p = p->next)
        f += pow(x,p->deg) * (p->coeff);
    return f;
}
```
poly *polySetCoeff(poly *q, int deg, double coeff) {
    if (!coeff) return q;
    polynode *p = q->head;
    if (!p || deg > p->deg) {
        // add to front
        polynode *r = malloc(sizeof(polynode));
        r->coeff = coeff;
        r->deg = deg;
        r->next = p;
        q->head = r;
        return q;
    }
    polynode *cur = q->head;
    for (; cur->next && cur->next->deg > deg; cur = cur->next);
    // More on next slide
if (cur->next && cur->next->deg == deg) {
    cur->next->coeff = coeff;
} else {
    polynode *r = malloc(sizeof(polynode));
    r->coeff = coeff;
    r->deg = deg;
    r->next = cur->next;
    cur->next = r;
}
return q;
int polyDegree (poly *p) {
    if (!p) return NEG_INF;
    return p->head->deg;
}
poly *polyCopy(poly *p) {
    poly *q = polyCreate();
    polynode *pnode = p->head
    while (pnode) {
        q = polySetCoeff(q, pnode->deg, pnode->coeff);
        pnode = pnode->next;
    }
    return q;
}
Generalized Lists

We discussed a linked list of integers, and one for polynomial terms.

What if we want a linked list of floats? Or a linked list of linked lists? Do we have to create a whole new structure with it’s own functions?

Ideally not, how can we create a linked list type that stores anything?
Instead of storing the data directly, our nodes can store pointers to the data. As we know a `void` pointer can point at anything, so we store `void` pointers in our nodes.

```c
typedef struct llnode {
    void *data;
    struct llnode *next;
} llnode;

typedef struct ll {
    llnode *head;
} ll;
```
Let’s write the following functions:

```c
// Creates an empty linked list.
ll *ll_create();

// Preconditions: elem is a pointer to heap allocated data, that this list now owns.
// Adds a new node storing elem at the front of our list.
void ll_addToFront(ll *l, void *elem);

// Preconditions: l is already sorted wrt to cmp.
// Adds a node storing elem to l in sorted order wrt to cmp.
void ll_addInOrder(ll *l, void *elem, int (*cmp)(void*, void*));
```
Generalized Lists

```c
ll *ll_create() {
    ll *ret = malloc(sizeof(ll));
    ret->head = NULL;
    return ret;
}
```
void ll_addToFront(ll *l, void *elem) {
  llnode *newNode = malloc(sizeof(llnode));
  newNode->data = elem;
  newNode->next = l->head;
  l->head = newNode;
}
// cmp returns:
// <0 if a should be before b
// ==0 if a == b
// >0 if a should be after b
void ll_addInOrder(ll *l, void *elem,
    int (*cmp)(void *a, void *b)) {
    llnode *prev = NULL;
    llnode *cur = l->head;
    // After this loop we want to insert elem
    // right after prev, right before cur.
    for(; cur && cmp(cur,cur->elem) > 0;
        prev = cur, cur = cur->next);
    // Check if we must update head
    if (!prev) addToFront(l, elem);
    // continued on next.
Generalized Lists

defined

else {
    llnode *n = malloc(sizeof(llnode));
    n->data = elem;
    n->next = cur;
    prev->next = n;
}
}
Deep Copying

Copying lists is a common thing to do. To make a deep copy we have to create a new node for each node in the list we’d like to copy. If the data is also pointers we might also like to deep copy those (rather than just copying pointers, so the nodes share data).

This means that to do so with our generalized list we need to take in a function from the user to copy the data.

```c
ll *ll_copy(ll *l, void *(*cpy)(void*));
```

It’s easiest to work with linked lists recursively though, so we’ll write a helper function that deep copys an individual node (and therefore also copies its next).
Deep Copying

```c
llnode *llnode_copy(llnode *n, 
    void *(*cpy)(void*)) {
    if (!n) return NULL;
    llnode *ret = malloc(sizeof(llnode));
    ret->next = llnode_copy(n->next, cpy);
    ret->data = cpy(n->data);
    return ret;
}

ll *ll_copy(ll *l, void *(*cpy)(void*)) {
    ll *ret = malloc(sizeof(ll));
    ll->head = llnode_copy(l->head, cpy);
    return ret;
}
```