CS 137 Week 10

Linked List

November 20th, 2017

This Week

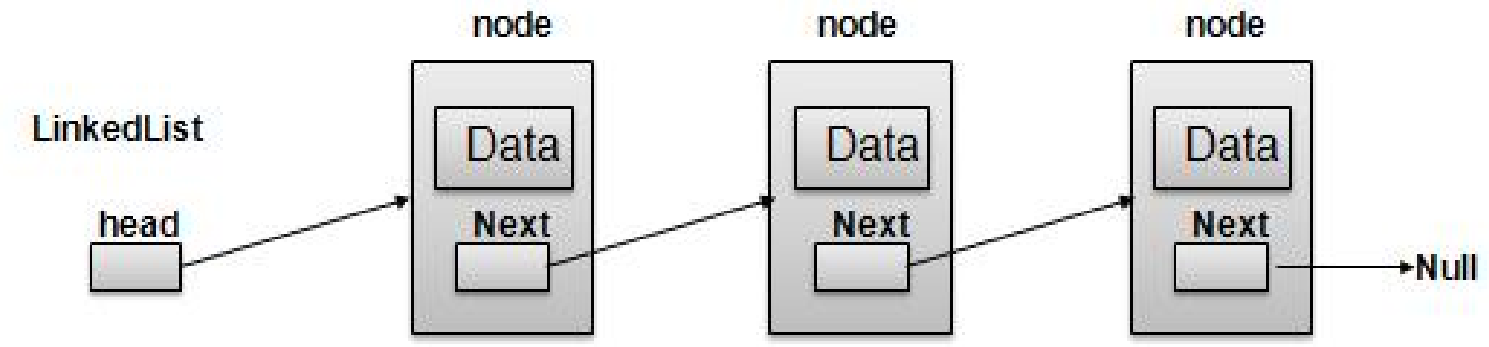
- This week we will introduce a complex data structure called a linked list.
- It is a structure where the data grows within it making it easy to insert new elements.
- Our primary example will be programming a polynomial

Linked List Framework

- A linked list consists of
  1. An item (I'll use an integer)
  2. A pointer to another Linked List element

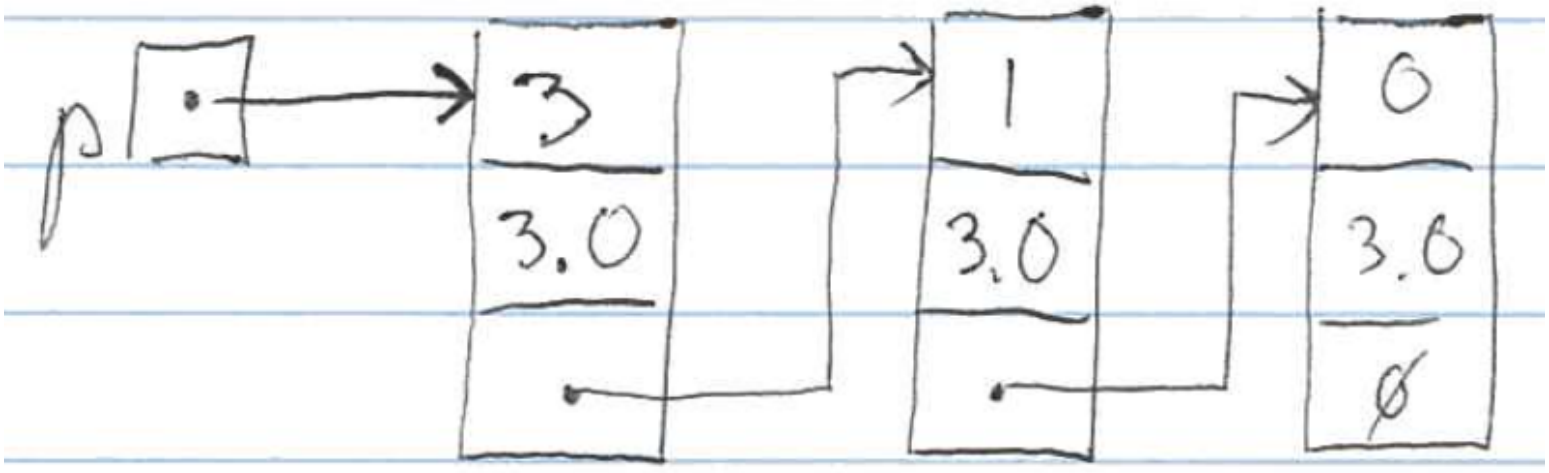
```
struct LL{
   int item
   struct LL next;
};
```

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 poly {
  double coeff;
  int deg;
  struct poly *next;
} 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);
```

One by One

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

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);

/*
Pre: poly *p is valid.
Post: compute x^{deg}p(1/x).
*/
poly *polyReverse(poly *p);
```

Implementation

```
poly *polyCreate() {
  return 0;
}
```

More Implementation

```
poly *polyDelete(poly *p) {
  while (p) {
    poly *t = p;
    p = p->next;
    free(t);
  }
  return p;
}
```

More Implementation

```
// Note p is passed *by value*
double polyEval(poly *p, double x) {
  double f = 0.0;
  // iterate over the nodes(terms) and evaluate e
  for (; p; p = p->next)
    f += pow(x,p->deg) * (p->coeff);
  return f;
}
```

More Implementation

```
poly *polySetCoeff(poly *p, int deg, double coeff
  if (!coeff) return p;
  if (!p || deg > p->deg) {
    poly *q = malloc(sizeof(poly));
    q->coeff = coeff;
    q->deg   = deg;
    q->next  = p;
    return q;
  }
  poly *cur = p;
  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 {
      poly *q = malloc(sizeof(poly));
      q->coeff = coeff;
      q->deg = deg;
      q->next = cur->next;
      cur->next = q;
    }
    return p;
}
```

More Implementation

```
int polyDegree (poly *p) {
  if (p == 0) return NEG_INF;
  return p->deg;
}
```

More Implementation

```
poly *polyCopy(poly *p){
  poly *q=polyCreate();
  while(p){
    q = polySetCoeff(q,p->deg,p->coeff);
    p = p->next;
  }

  return q;
}
```

More Implementation

```
poly *polyReverse(poly *p) {
  poly *prev = 0;
  poly *cur = polyCopy(p);
  poly *next = 0;
  while(cur) {
    next = cur->next;
    cur->next = prev;
    prev = cur;
    cur = next;
  }
  return prev;
}
```