

Linked List

Polynomials

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024



Outline

- 1 Polynomial Representation
- 2 Additional List Operations



Outline

- 1 Polynomial Representation
- 2 Additional List Operations



Goal

Represent the polynomial:

$$a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \cdots + a_0x^{e_0}.$$



Goal

Represent the polynomial:

$$a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \cdots + a_0x^{e_0}.$$

- **Idea:** Represent each term as a **node** containing
 - **coefficient** field
 - **exponent** field
 - **pointer** to the next term



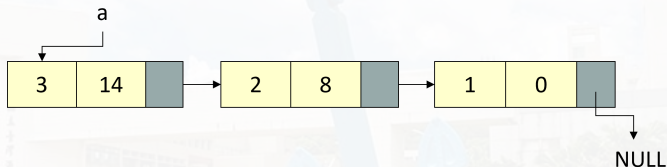
Declaration

```
typedef struct polyNode *polyPointer;
struct polyNode {
    int coef;
    int expon;
    polyPointer link;
};
polyPointer a, b;
```

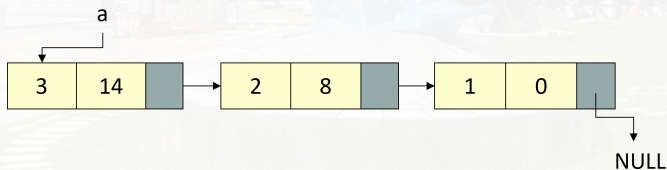


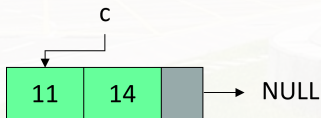
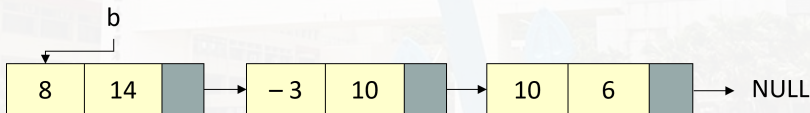
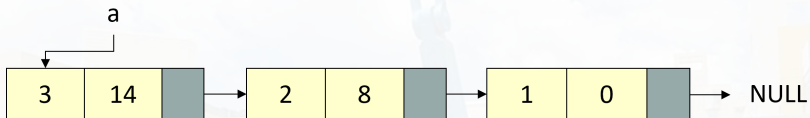
Examples

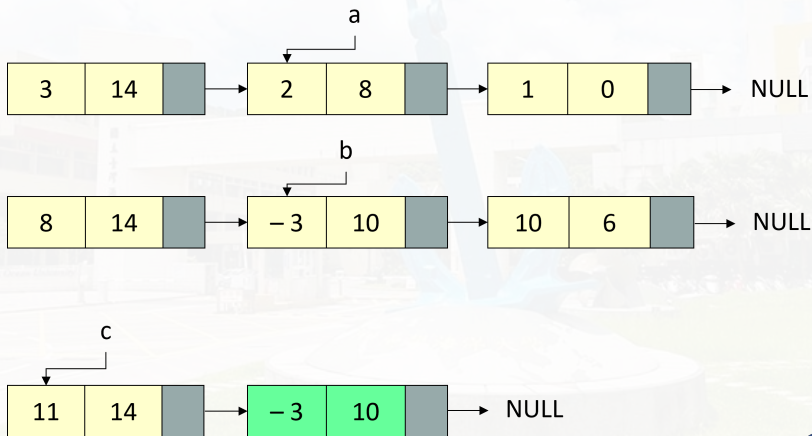
- $a = 3x^{14} + 2x^8 + 1$

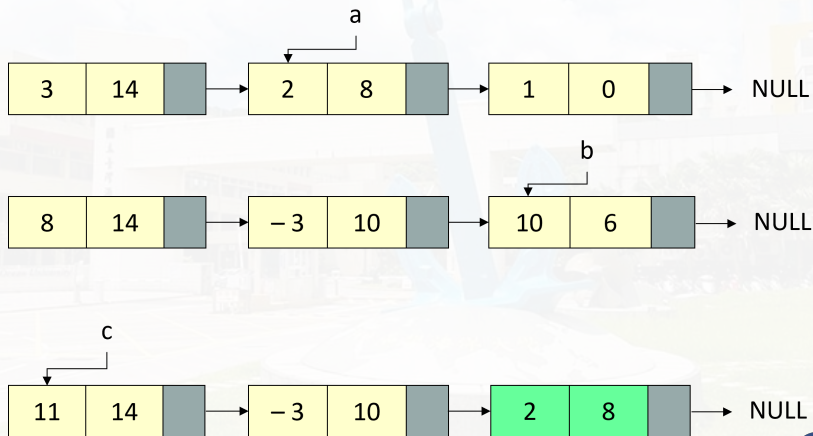


- $b = 8x^{14} - 3x^{10} + 10x^6$



Generating the first three terms of $c = a + b$ 

Generating the first three terms of $c = a + b$ 

Generating the first three terms of $c = a + b$ 

Addition of Two Polynomials

```

polyPointer polyAdd(polyPointer a, polyPointer b) { /* return a polynomial which is the sum of a and b */
    polyPointer front, rear, temp;
    int sum; rear = (polyPointer)malloc(sizeof(*rear));
    if (IS_FULL(rear)) { printf("The memory is full\n"); exit(1) }
    front = rear;
    while (a && b) {
        switch (COMPARE(a->expon, b->expon)) {
            case -1: /* a->expon < b->expon */
                attach(b->coef, b->expon, &rear);
                b = b->link;
                break;
            case 0: /* a->expon = b->expon */
                sum = a->coef + b->coef;
                if (sum)
                    attach(sum, a->expon, &rear);
                a = a->link; b = b->link; break;
            case 1: /* a->expon > b->expon */
                attach(a->coef, a->expon, &rear);
                a = a->link;
        }
        /* copy rest of list a and then list b*/
        for (; a; a = a->link) attach(a->coef, a->expon, &rear);
        for (; b; b = b->link) attach(b->coef, b->expon, &rear);
        rear->link = NULL;
        /* delete extra initial node */
        temp = front; front = front->link; free(temp);
        return front;
    }
}

```

Attach a new node to the end of a list

```
void attach(float coefficient, int exponent, polyPointer *ptr) {  
    /* create a new node with coef = coefficient and expon = exponent,  
    attach it to the node pointed by ptr and update ptr to point to this new node */  
    polyPointer temp;  
    temp = (polyPointer)malloc(sizeof(*temp));  
    if (IS_FULL(temp)) {  
        printf("The memory is full\n");  
        exit(1);  
    }  
    temp->coef = coefficient;  
    temp->expon = exponent;  
    (*ptr)->link = temp;  
    *ptr = temp;  
}
```



Analysis of polyAdd

Consider

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \cdots + a_0x^{e_0},$$

$$B(x) = b_{m-1}x^{f_{m-1}} + b_{m-2}x^{f_{m-2}} + \cdots + b_0x^{f_0}$$



Analysis of polyAdd

Consider

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \cdots + a_0x^{e_0},$$

$$B(x) = b_{m-1}x^{f_{m-1}} + b_{m-2}x^{f_{m-2}} + \cdots + b_0x^{f_0}$$

- coefficient additions:

$$0 \leq \# \text{coefficient additions} \leq \min\{m, n\}$$



Analysis of polyAdd

Consider

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \dots + a_0x^{e_0},$$
$$B(x) = b_{m-1}x^{f_{m-1}} + b_{m-2}x^{f_{m-2}} + \dots + b_0x^{f_0}$$

- coefficient additions:

$$0 \leq \# \text{coefficient additions} \leq \min\{m, n\}$$

- exponent comparisons:

- In each iteration, either pointer a or b or both move to the next term(s).
- The maximum number of exponent comparisons is $m + n$

- create new nodes for C :



Analysis of polyAdd

Consider

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \dots + a_0x^{e_0},$$
$$B(x) = b_{m-1}x^{f_{m-1}} + b_{m-2}x^{f_{m-2}} + \dots + b_0x^{f_0}$$

- coefficient additions:

$$0 \leq \# \text{coefficient additions} \leq \min\{m, n\}$$

- exponent comparisons:

- In each iteration, either pointer a or b or both move to the next term(s).
- The maximum number of exponent comparisons is $m + n$

- create new nodes for C :

- The maximum number of terms in C is



Analysis of polyAdd

Consider

$$A(x) = a_{m-1}x^{e_{m-1}} + a_{m-2}x^{e_{m-2}} + \dots + a_0x^{e_0},$$
$$B(x) = b_{m-1}x^{f_{m-1}} + b_{m-2}x^{f_{m-2}} + \dots + b_0x^{f_0}$$

- coefficient additions:

$$0 \leq \# \text{coefficient additions} \leq \min\{m, n\}$$

- exponent comparisons:

- In each iteration, either pointer a or b or both move to the next term(s).
- The maximum number of exponent comparisons is $m + n$

- create new nodes for C :

- The maximum number of terms in C is $O(m + n)$.



Erasing Polynomials

- One by one, free the nodes pointed by ptr.

```
void erase(polyPointer *ptr) {  
    /* erase the polynomial pointed to by ptr */  
    polyPointer temp;  
    while (*ptr) {  
        temp = *ptr;  
        *ptr = (*ptr)->link;  
        free(temp);  
    }  
}
```



A Question

Can we free all the nodes of a polynomial more efficiently?



A Question

Can we free all the nodes of a polynomial more efficiently?

- Consider the **circular list representation**.

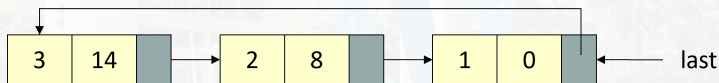


Circular List Representation

Circular List

A singly linked list in which the link field of the last node points to the first node is called a circular list.

$$3x^{14} + 2x^8 + 1:$$



- In order to obtain an efficient erase algorithm, we maintain a list of nodes that have been **freed**.

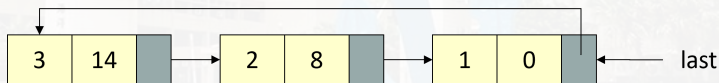


Circular List Representation

Circular List

A singly linked list in which the link field of the last node points to the first node is called a circular list.

$$3x^{14} + 2x^8 + 1:$$



- In order to obtain an efficient erase algorithm, we maintain a list of nodes that have been **freed**.
- Only when the list is empty do we need to use `malloc` to create a new node.



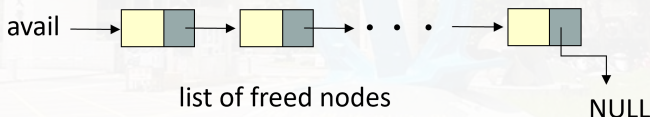
Functions to deal with free nodes of polynomials

- Let `avail` be a variable of type `polyPointer` which points to the `first` node in the list of freed nodes.



Functions to deal with free nodes of polynomials

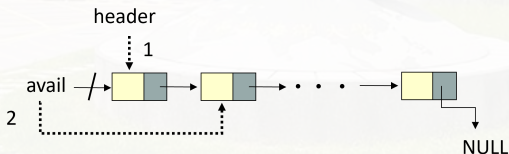
- Let `avail` be a variable of type `polyPointer` which points to the `first` node in the list of freed nodes.
- Instead of using `malloc()` and `free()`, we now use `getNode()` and `retNode()`.



getNode()

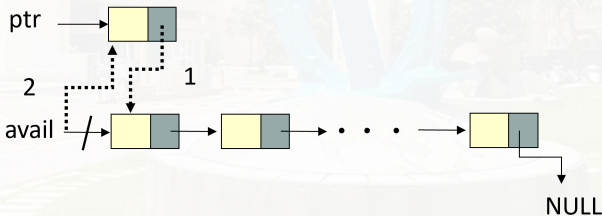
- ★ To avoid the case of zero polynomials, we introduce a header into each polynomial.

```
polyPointer getNode(void) {  
    /* provide a node header for use */  
    polyPointer header;  
    if (avail) {  
        header = avail;  
        avail = avail->link;  
    } else  
        malloc(header, sizeof(*header));  
    return header;  
}
```



retNode()

```
void retNode(polyPointer ptr) {  
    /* return a node to the available list */  
    ptr->link = avail;  
    avail = ptr;  
}
```

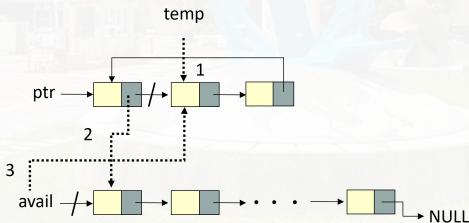


Erasing a circular list

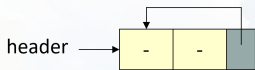
```

void cir_erase(polyPointer *ptr) {
    polyPointer temp;
    if (*ptr) {
        temp = (*ptr)->link;
        (*ptr)->link = avail;
        avail = temp;
        *ptr = NULL;
    }
}

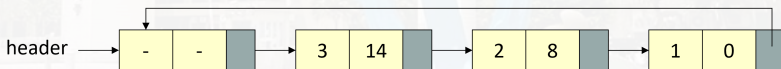
```



Polynomials with header nodes (Examples)



(a) Zero polynomial

(b) $3x^{14} + 2x^8 + 1$

Outline

- 1 Polynomial Representation
- 2 Additional List Operations**



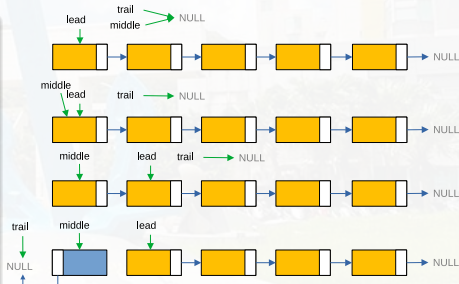
Additional List Operations

- a chain (singly linked list in which the **last** node has a **null link**):
 - inversion (反轉)
 - concatenation (連接) of two chains
- a circular list (singly linked list in which the **last** node **points to the first node**):
 - insert a node at the front of a circular list
 - determine the length of a circular list



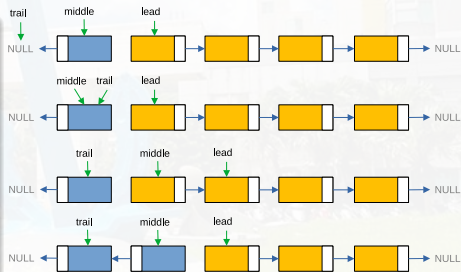
Inversion of a chain

```
listPointer invert (listPointer lead) {
  // invert the list pointed to by lead
  listPointer middle, trail;
  middle = NULL;
  while (lead) {
    trail = middle;
    middle = lead;
    lead = lead->link;
    middle->link = trail;
  }
  return middle;
}
```



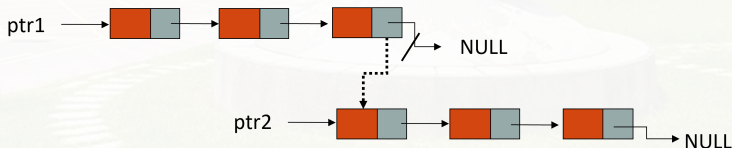
Inversion of a chain

```
listPointer invert (listPointer lead) {
// invert the list pointed to by lead
    listPointer middle, trail;
    middle = NULL;
    while (lead) {
        trail = middle;
        middle = lead;
        lead = lead->link;
        middle->link = trail;
    }
    return middle;
}
```



Concatenation of two singly linked lists

```
listPointer concatenate (listPointer ptr1, listPointer ptr2) {
  /* produce a new list that contains the list
  ptr1 followed by the list ptr2. */
  listPointer temp;
  // check for empty lists
  if (!ptr1) return ptr2;
  if (!ptr2) return ptr1;
  /* neither list is empty, find end of first list */
  for (temp = ptr1; temp->link; temp = temp->link);
  // linked end of first to start of second
  temp->link = ptr2;
}
```

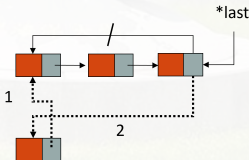
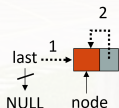


Inserting at the front of a circular list

```

void insertFront (listPointer *last, listPointer node) {
  /* insert node at the front of the circular list
   whose last node is last */
  if (!(*last)){
    // list is empty, change last to point to new entry
    *last = node;
    node->link = node;
  } else {
    // list is not empty, add a new entry at front
    node->link = (*last)->link;
    (*last)->link = node;
  }
}

```



Find the length of a **circular list**

```
int length (listPointer last) {  
    /* find the length of circular list last */  
    listPointer temp;  
    int count = 0;  
    if (last) {  
        temp = last;  
        do {  
            count++;  
            temp = temp->link;  
        } while (temp != last);  
    }  
    return count;  
}
```



Discussions

