Pointers and Lists

CSE 326

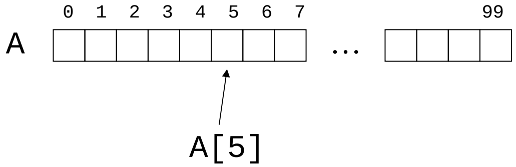
Data Structures

Unit 2

Reading: Section 3.2 The List ADT

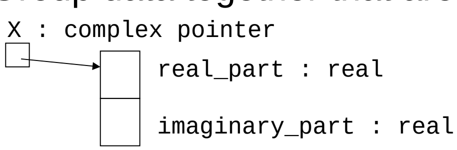
Basic Types and Arrays

- Basic Types
  - › integer, real (floating point), boolean (0,1), character
- Arrays
  - › A[0..99] : integer array



Records and Pointers

- Record (also called a struct)
  - › Group data together that are related



  - › To access the fields we use “dot” notation.

```
X.real_part
X.imaginary_part
```


Record Definition

- Record definition creates a new type

Definition

```
record complex : (
  real_part : real,
  imaginary_part : real
)
```

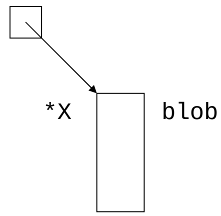
Use in a declaration

```
X : complex
```

Pointer

- A pointer is a reference to a variable or record (or object in Java world).

```
X : blob pointer
```

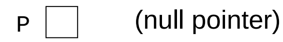


- In C, if X is of type pointer to Y then *X is of type Y

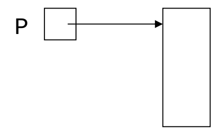
Creating a Record

- We use the “new” operator to create a record.

  P : pointer to blob;



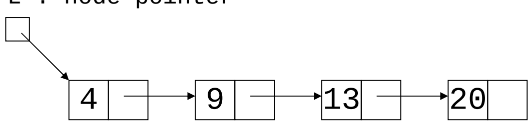
  P := new blob;



Simple Linked List

- A linked list
  - Group data together in a flexible, dynamic way.
  - We’ll describe several list ADTs later.

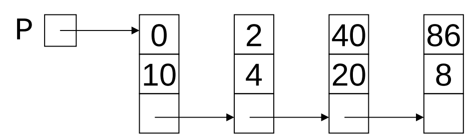
```
L : node pointer
```




```
record node : (
  data : integer,
  next : node pointer
)
```

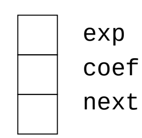

Application Sparse Polynomials

- $10 + 4x^2 + 20x^{40} + 8x^{86}$

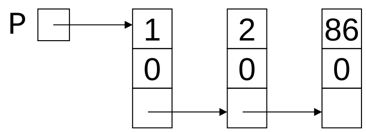


Exponents in Increasing order

```
record poly : (
  exp : integer,
  coef : integer,
  next : poly pointer
)
```

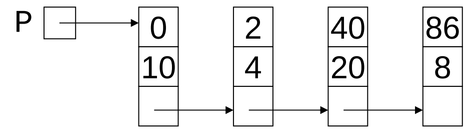


Identically Zero Polynomial

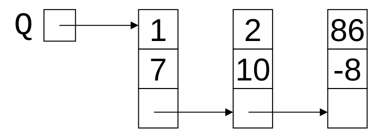


Addition of Polynomials

$10 + 4\,x^{2} + 20\,x^{40} + 8\,x^{86}$



$7\,x + 10\,x^{2} - 8\,x^{86}$

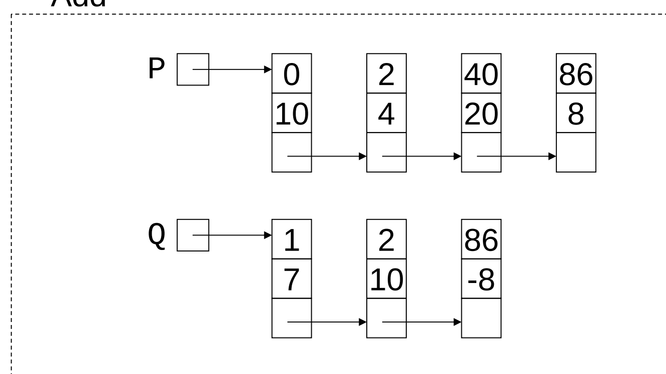


Recursive Addition

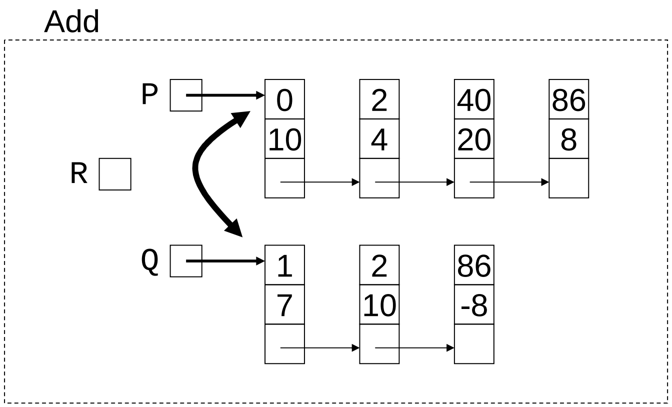
```
Add(P, Q : poly pointer): poly pointer{
R : poly pointer
    case {
      P = null : R := Q ;
      Q = null : R := P ;
      P.exp < Q.exp : R := P ;
                      R.next := Add(P.next,Q);
      P.exp > Q.exp : R := Q ;
                      R.next := Add(P,Q.next);
      P.exp = Q.exp : R := P ;
                      R.coef := P.coef + Q.coef ;
                      R.next := Add(P.next,Q.next);
    }
     return R
}
```


Example

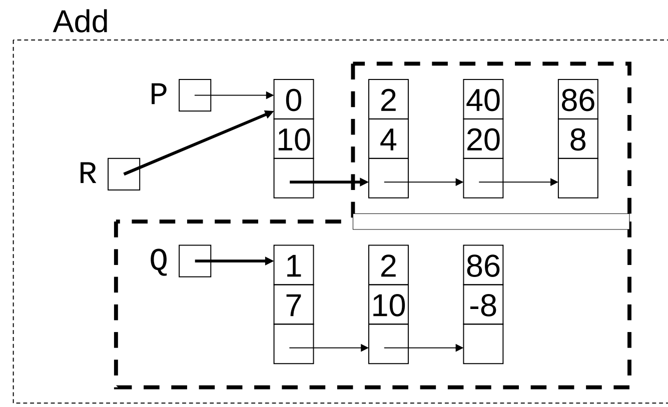
Add



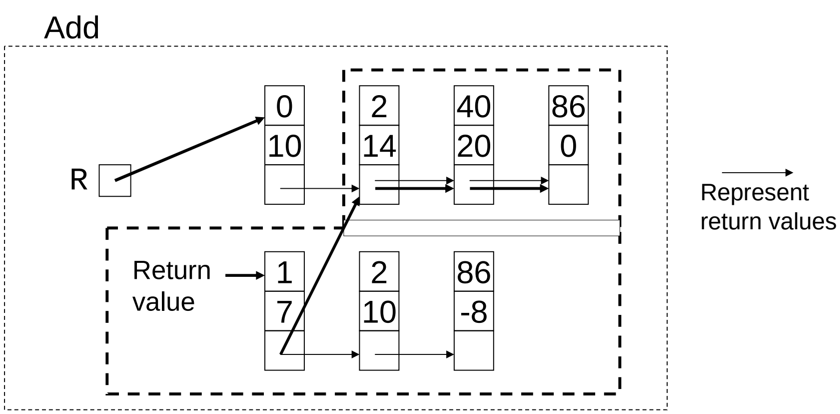
Example (first call)



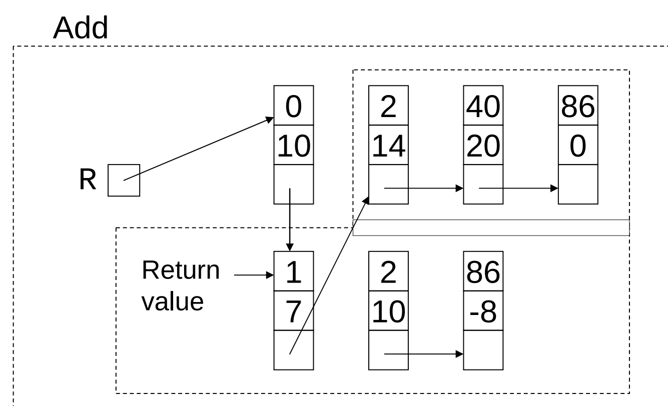
The Recursive Call



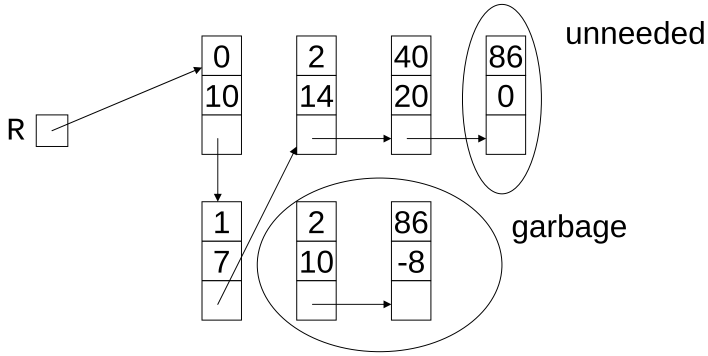
During the Recursive Call



After the Recursive Call



The final picture



Notes on Addition

- Addition is destructive, that is, the original polynomials are gone after the operation.
- We don't salvage "garbage" nodes. Let's talk about this.
- We don't consider the case when the coefficients cancel. Let's talk about this.

Unneeded nodes to Garbage

- How would you force the unneeded node to be garbage in the code on slide 11?
- Suggestions?

Memory Management – Global Allocator

- Global Allocator's store – always get and return blocks to global allocator – an area in the memory from which we can dynamically allocate memory.
  - The user (the program) must 'free' the memory when done.

Memory Management – Garbage Collection

- Garbage collection – run time system recovers inaccessible blocks from time-to-time. Used in Lisp, Smalltalk, Java.
  - + No need to return blocks to an allocator.
  - - Care must be taken to make unneeded blocks inaccessible.
  - - When garbage collection kicks in there may be undesirable response time.

Solution for Polyn. Addition

```
P.exp = Q.exp : R := P ;
                R.coef := P.coef + Q.coef ;
                if R.coef = 0 then
                    R := Add(P.next,Q.next);
// The terms with coef = 0 have been removed from the
// result
                else
                    R.next := Add(P.next,Q.next);
}
```


Use of Global Allocator

```
P.exp = Q.exp : R := P ;
                R.coef := P.coef + Q.coef ;
                if R.coef = 0 then
                    R := Add(P.next,Q.next);
                    Free(P); Free(Q);
                else
                    R.next := Add(P.next,Q.next);
                    Free(Q);
}
```


List ADT

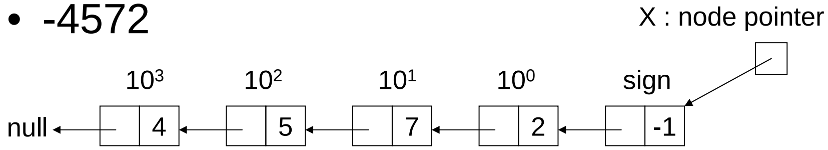
- What is a List?
  - › Ordered sequence of elements A_1, A_2, …, A_N
- Elements may be of arbitrary type, but all are of the same type
- Common List operations are:
  - › Insert, Find, Delete, IsEmpty, IsLast, FindPrevious, First, Kth, Last, Print, etc.

Simple Examples of List Use

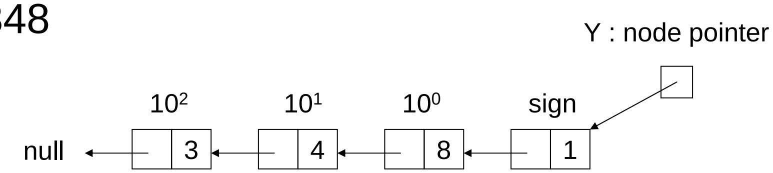
- Polynomials
  - › $25 + 4x^2 + 75x^{85}$
- Unbounded Integers
  - › 4576809099383658390187457649494578
- Text
  - › “This is an example of text”

Unbounded Integers Base 10

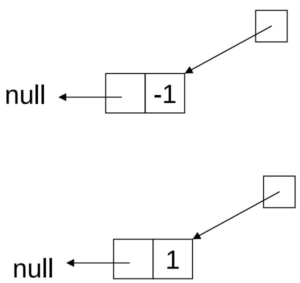
- -4572



- 348



Zero



List Implementations

- Two types of implementation:
  - › Array-Based
  - › Pointer-Based

List: Array Implementation

- Basic Idea:
  - › Pre-allocate a big array of size MAX_SIZE
  - › Keep track of current size using a variable **`count`**
  - › Shift elements when you have to insert or delete

0	1	2	3	…	count-1		MAX_SIZE-1
A_1	A_2	A_3	A_4	…	A_N		

List: Array Implementation

Insert Z in 3rd position

0	1	2	3	4	5			MAX_SIZE-1
A	B	C	D	E	F			

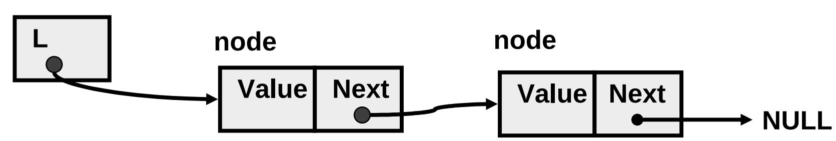
0	1	2	3	4	5	6		MAX_SIZE-1
A	B	Z	C	D	E	F		

Array List Insert Running Time

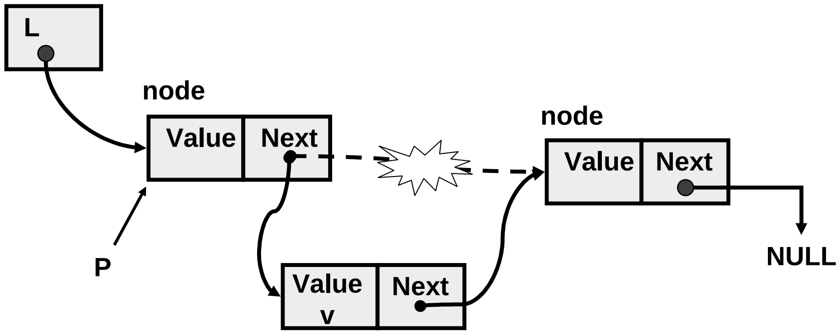
- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other hand – we can access the kth item in O(1).

List: Pointer Implementation

- Basic Idea:
  - › Allocate little blocks of memory (nodes) as elements are added to the list
  - › Keep track of list by linking the nodes together
  - › Change links when you want to insert or delete



Pointer-based Insert (after p)



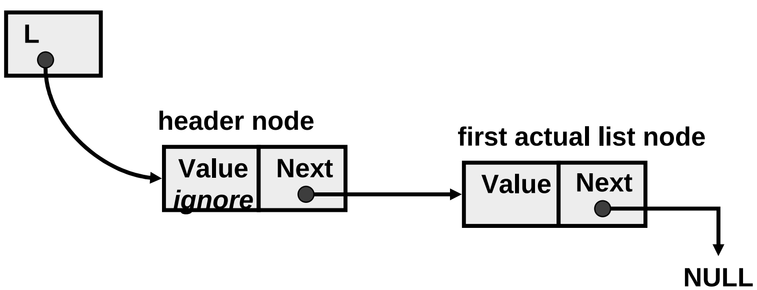
Insert the value v after P

Insertion After

```
InsertAfter(p : node pointer, v : value_type): {
   x : node pointer;
   x := new node;
   x.value := v;
   x.next := p.next;
   p.next := x;
}
```

Note: cannot swap two last lines (why?)

Linked List with Header Node



Advantage: “insert after” and “delete after” can be done at the beginning of the list.

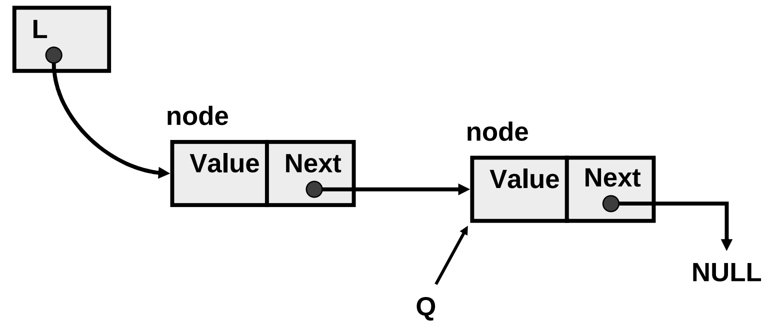
Pointer Implementation Issues

- Whenever you break a list, your code should fix the list up as soon as possible
  - › Draw pictures of the list to visualize what needs to be done
- Pay special attention to boundary conditions:
  - › Empty list
  - › Single item – same item is both first and last
  - › Two items – first, last, but no middle items
  - › Three or more items – first, last, and middle items

Pointer List Insert Running Time

- Running time for a list with N elements?
- Insert takes constant time (O(1))
- Does not depend on list size
- Compare to array based list which is O(N)

Linked List Delete



To delete the node pointed to by Q, need a pointer to the previous node; See book for findPrevious method

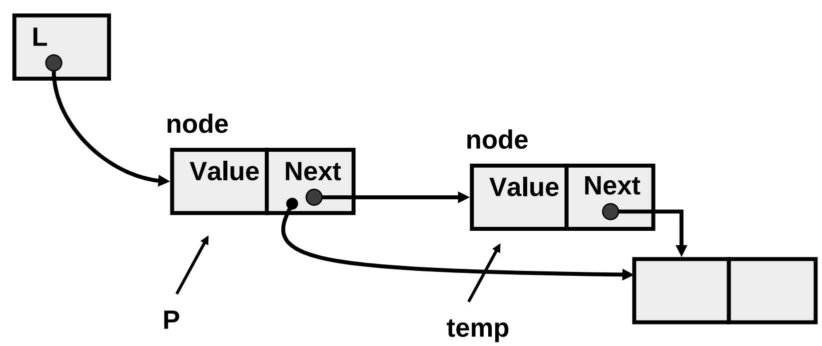
Delete After

```
DeleteAfter(p : node pointer): {
      temp : node pointer;
      temp = p.next;
      p.next = temp.next; //p.next.next
      free(temp);
}
```

Note: p points to the node that comes before the deleted node!

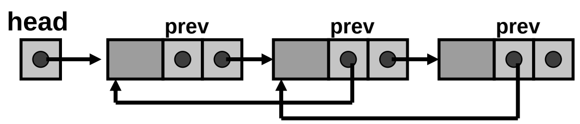
temp – the node to be removed.

Linked List Delete



Doubly Linked Lists

- findPrevious (and hence Delete) is slow [O(N)] because we cannot go directly to previous node
- Solution: Keep a "previous" pointer at each node

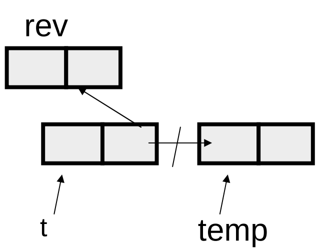


Double Link Pros and Cons

- Advantage
  - Delete (not DeleteAfter) and FindPrev are faster
- Disadvantages:
  - More space used up (double the number of pointers at each node)
  - More book-keeping for updating the two pointers at each node (pretty negligible overhead)

Reverse a linked list

```
Reverse(t : node pointer): node pointer  {
   rev : node pointer;
   temp: node pointer;
   rev = NULL;
   while(t !=NULL){
     temp = t.next;
     t.next = rev;
     rev = t;
     t = temp;
   }
   return (rev);
}
```



rev: the ‘already reversed’ part.

Why do we need temp?

Implementing Pointers in Arrays – “Cursor Implementation”

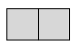
- This is needed in languages like Fortran, Basic, and assembly language
- Easiest when number of records is known ahead of time.
- Each record field of a basic type is associated with an array.
- A pointer field is an unsigned integer indicating an array index.

Idea

Pointer World

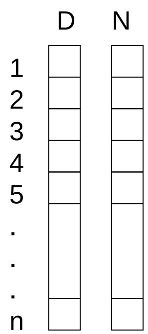
n nodes

data next



data : basic type
next : node pointer

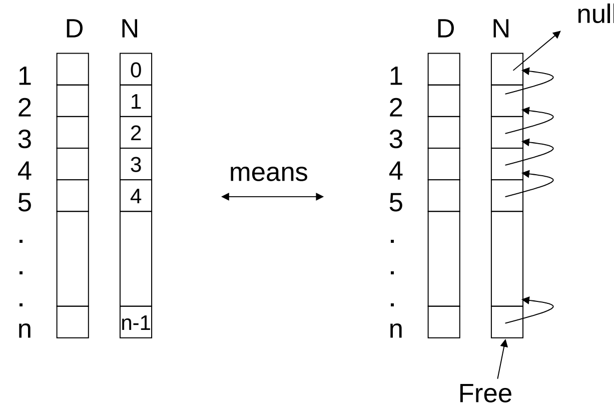
Nonpointer World



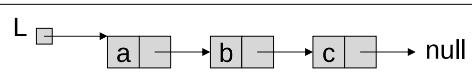
- D[ ] : basic type array
- N[ ] : integer array
- Pointer is an integer
- null is 0
- p.data is D[p]
- p.next is N[p]
- Free list needed for node allocation

Initialization

Free = n



Example of Use



n = 8
L = 4
Free = 7

	D	N
1		3
2	c	0
3		0
4	a	6
5		8
6	b	2
7		5
8		1

```
InsertFront(L : integer, x : basic type) {
    q : integer;
    if not(Free = 0) then q := Free
      else return “overflow”;
    Free := N[Free];
    D[q] := x;
    N[q] := L;
    L := q;
}
```


Try DeleteFront

- Define the cursor implementation of DeleteFront which removes the first member of the list when there is one.
  - › Remember to add garbage to free list.

```
DeleteFront(L : integer) {
???
}
```

DeleteFront Solution

```
DeleteFront(L : integer) {
   q : integer;
   if L = 0 then return “underflow”
   else {
     q := L;
     L := N[L];
     N[q] := Free;
     Free := q;
   }
}
```