Lists

CSE 373

Data Structures

Unit 3

Lists

- Reading
  - › Section 3.1 ADT (recall, lecture 1):
    - Abstract Data Type (ADT): Mathematical description of an object with set of operations on the object.
  - › Section 3.2 The List ADT

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”

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			

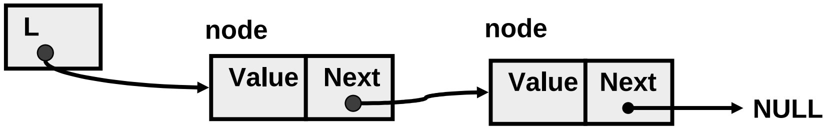
0	1	2	3	4	5	6		MAX_SIZE-1
A	B	Z	C	D	E	F		

Array List Insert Running Time

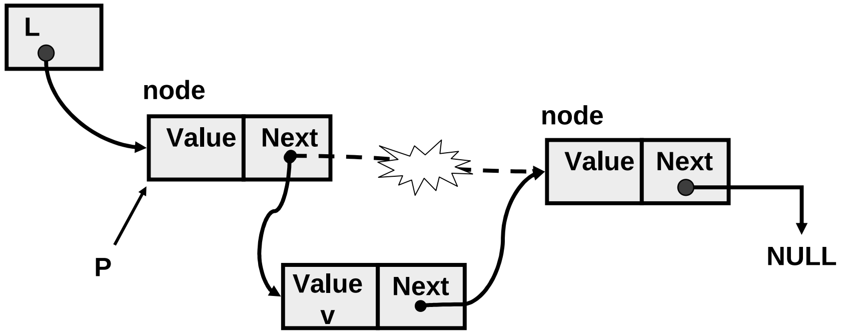
- Running time for a list with N elements?
- On average, must move half the elements to make room – assuming insertions at positions are equally likely
- Worst case is insert at position 0. Must move all N items one position before the insert
- This is O(N) running time. Probably too slow
- On the other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



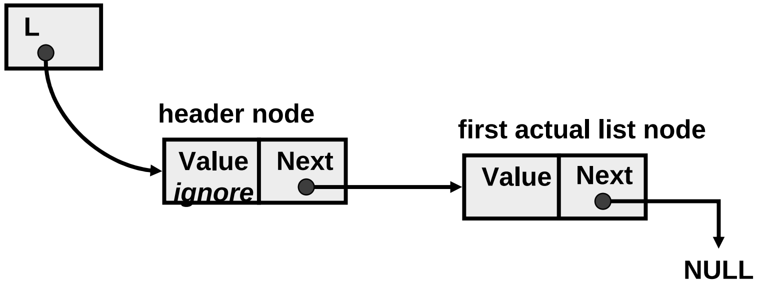
Insert the value v after P

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.

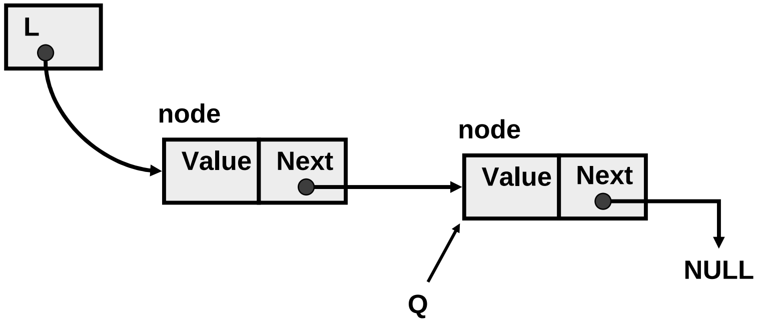
Pointer Implementation Issues

- 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

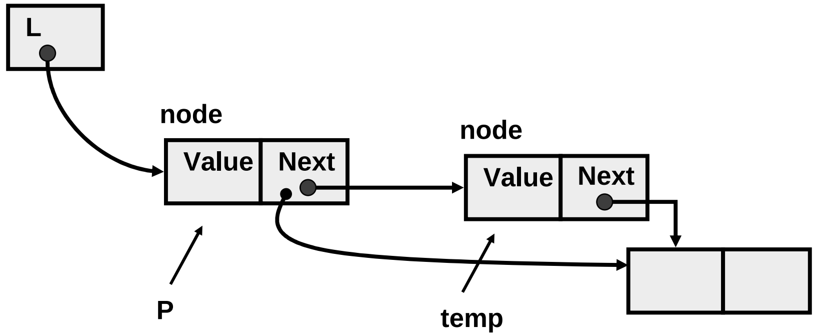
Delete After

```
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!

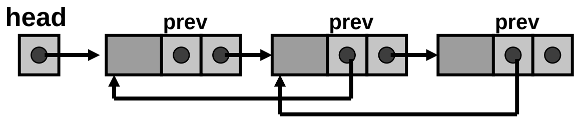
temp – the node to be removed.

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

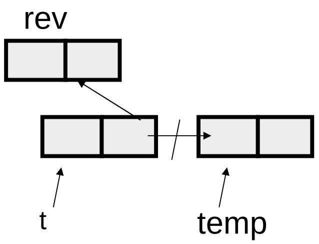


Double Link Pros and Cons

- 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);
}
```

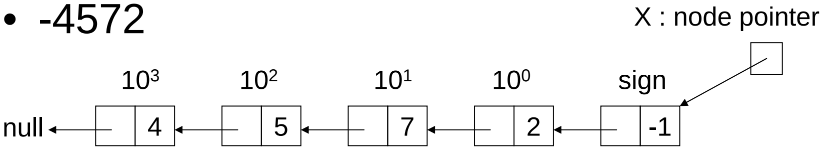


rev: the ‘already reversed’ part.

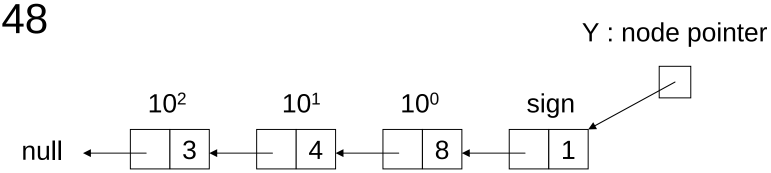
Why do we need temp?

Unbounded Integers Base 10

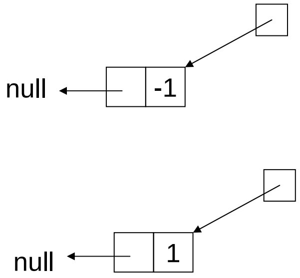
- -4572



- 348

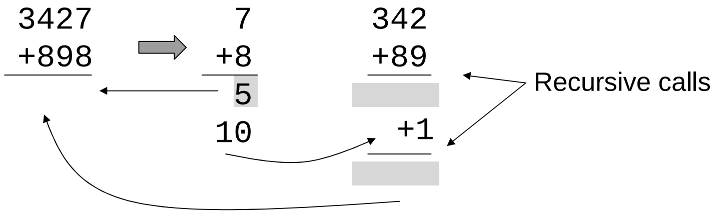


Zero



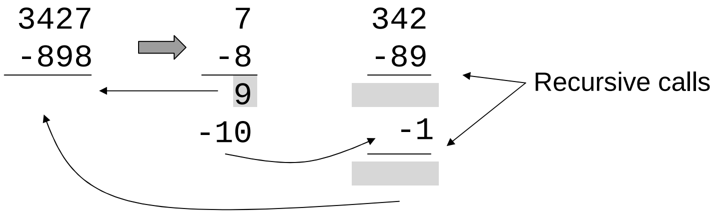
Recursive Addition

- Positive numbers (or negative numbers)



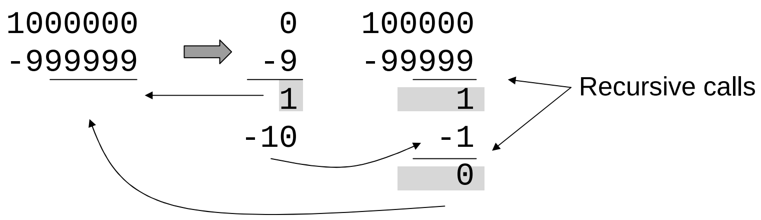
Recursive Addition

- Mixed numbers



Example

- Mixed numbers



1000000
-999999

0
-9
1
-10

100000
-99999
1
-1
0

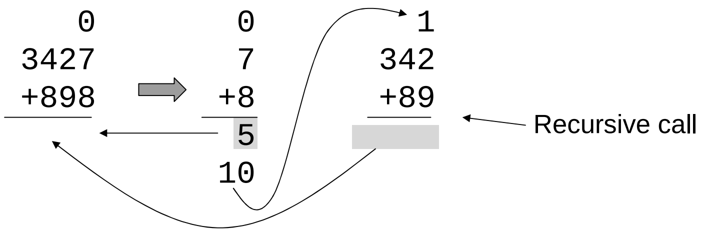
Recursive calls

Alternative Addition

- Use an auxiliary function
  - AddAux(p,q : node pointer, cb : integer) which returns the result of adding p and q and the carry/borrow cb.
  - Add(p,q) := AddAux(p,q,0)
  - Advantage: more like what we learned in school (and more like actual binary adders in hardware).

Auxiliary Addition

- Positive numbers



0
3427
+898

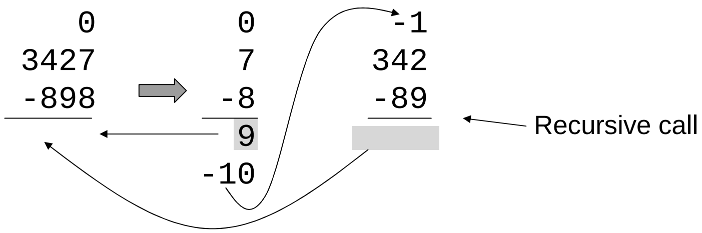
0
7
+8
5
10

1
342
+89

Recursive call

Auxiliary Addition

- Mixed numbers



0
3427
-898

0
7
-8
9
-10

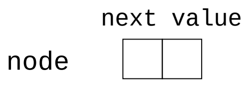
-1
342
-89

Recursive call

Copy

- Design a recursive algorithm to make a copy of a linked list (like the one used for long integers)

```
Copy(p : node pointer) : node pointer {
???
}
```



next value

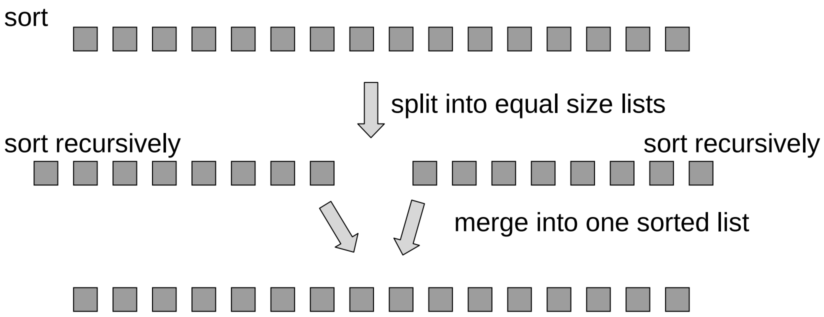
node

Comparing Integers

```
IsZero(p : node pointer) : boolean { //p points to the sign node
    return p.next = null;
}
IsPositive(p: node pointer) : boolean {//p points to the sign node
    return not IsZero(p) and p.value = 1;
}
Negate(p : node pointer) : node pointer {  //destructive
    if p.value = 1 then p.value := -1
    else p.value := 1;
    return p;
}
LessThan(p,q :node pointer) : boolean { // non destructive
    p1,q1 : node pointer;
    p1 := Copy(p); q1 := Copy(q);
    return IsPositive(Add(q1,Negate(p1)); // x < y iff 0 < y – x
      //We assume Add and Negate are destructive
}
```

List Mergesort

- Overall sorting plan



sort

split into equal size lists

sort recursively

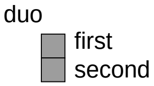
sort recursively

merge into one sorted list

Mergesort pseudocode

```
Mergesort(p : node pointer) : node pointer {
Case {
  p = null : return p; //no elements
  p.next = null : return p; //one element
  else
    d : duo pointer; // duo has two fields first,second
    d := Split(p);
    return Merge(Mergesort(d.first),Mergesort(d.second));
}
}
```

Note: Mergesort is destructive.



duo

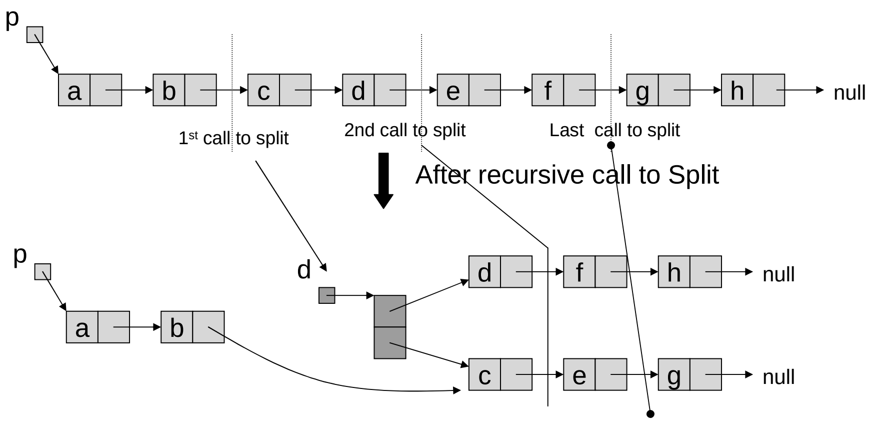
first

second

Split

```
Split(p : node pointer) : duo pointer {
d : duo pointer;
Case {
  p = null : d := new duo; return d//both fields are null
  p.next = null : d := new duo; d.first := p ; return d
            //d.second is null
  else :
    d := Split(p.next.next);
    p.next.next := d.first;
    d.first := p.next;
    p.next := d.second;
    d.second := p;
    return d;
}
}
```

Split Example



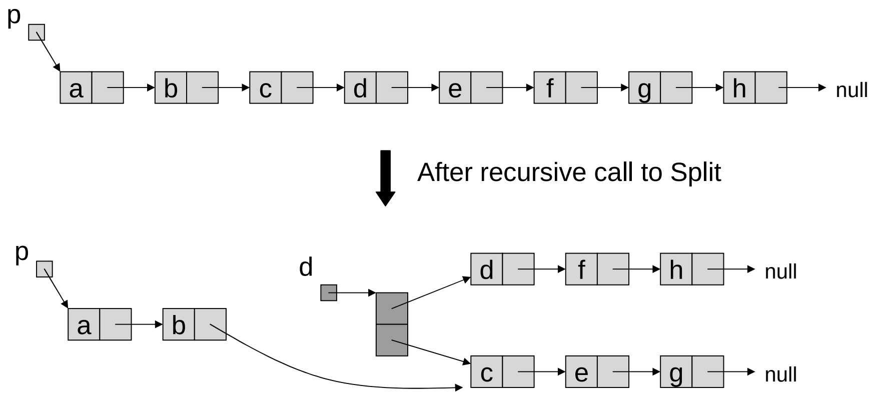
1st call to split

2nd call to split

Last call to split

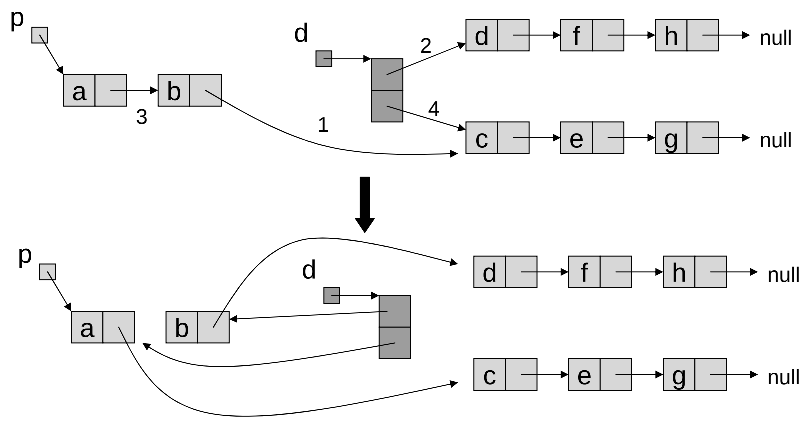
After recursive call to Split

Split Example



After recursive call to Split

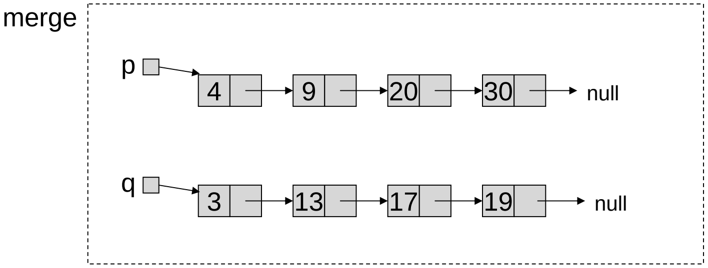
Split Example



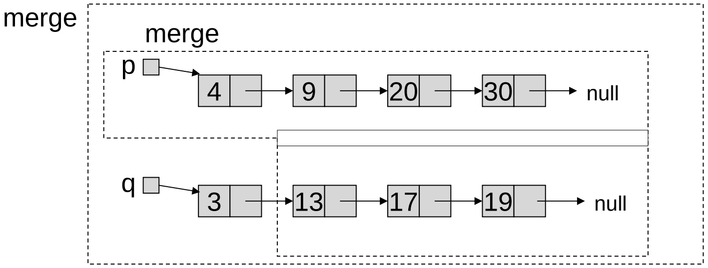
Merge

```
Merge(p,q : node pointer): node pointer{
   case {
     p = null : return q;
     q = null : return p;
     LessThan(p.value,q.value) :
       p.next := Merge(p.next,q);
       return p;
     else :
       q.next := Merge(p,q.next);
       return q;
   }
}
```

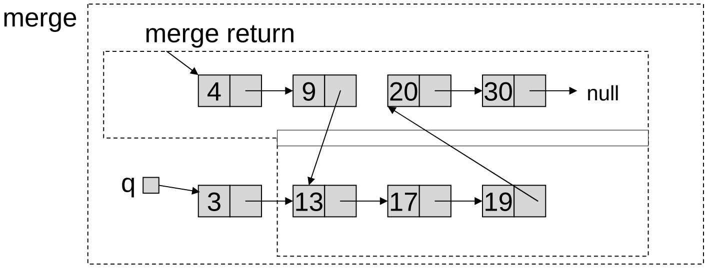
Merge Example



Merge Example



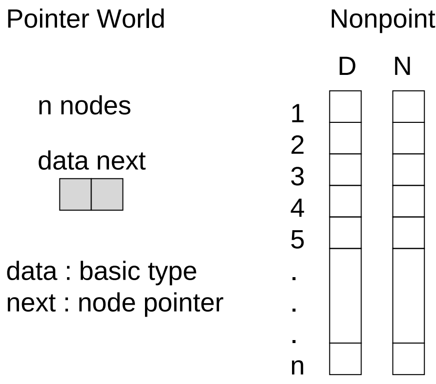
Merge Example



Implementing Pointers in Arrays – “Cursor Implementation”

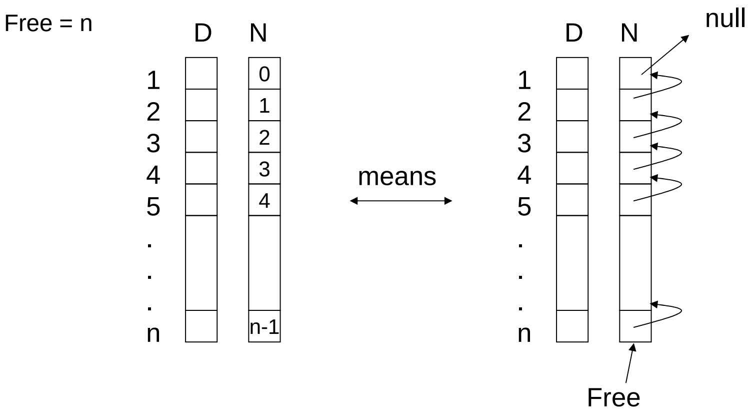
- This is needed in languages like Fortran, Basic, and assembly language
- Easiest when number of records is known ahead of time.
- Each record field of a basic type is associated with an array.
- A pointer field is an unsigned integer indicating an array index.

Idea

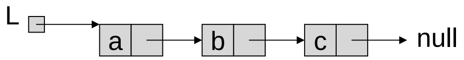


- D[ ] : basic type array
- N[ ] : integer array
- Pointer is an integer
- null is 0
- p.data is D[p]
- p.next is N[p]
- Free list needed for node allocation

Initialization



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) {
???
}
```

Copy Solution

```
Copy(p : node pointer) : node pointer {
   if p = null then return null
   else {
     q : node pointer;
     q := new node; //by convention the value
                    //field is 0 and the
                    //pointer field is null
     q.value := p.value;
     q.next := Copy(p.next);
     return q;
   }
}
```

DeleteFront Solution

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