数据结构与算法分析

华中科技大学软件学院

2014年秋

大纲

课程计划

- 已经学习了
  - 时间复杂度的表示
  - 算法的分析方法
  - 递归与迭代的转换
  - 设计算法以改善时间复杂度

课程计划

- 已经学习了
  - 时间复杂度的表示
  - 算法的分析方法
  - 递归与迭代的转换
  - 设计算法以改善时间复杂度
- 即将学习
  - 抽象数据结构：线性表
  - 数组与链表
  - 桶式排序与基数排序
  - 堆栈及其应用
  - 队列

Roadmap

抽象数据结构

- 通常一种数据结构和某些特殊的算法关系密切
  - 可以把数据结构视为OOP中的对象
- 数据结构的选择依赖于所希望执行的算法
  - Sorted Array: selection, search: fast, but insertion, deletion: slow
  - Linked list: insertion, deletion-at-position: fast, selection, search: slow

线性表

- 最简单的一种数据结构：线性表
- 元素之间保持线性顺序：最多一个直接前驱，最多一个直接后继
- 定义在线性表上的操作
  - Insert
  - Delete
  - Sort
  - Retrieve
- 可以将线性表看做一个提供这些操作的“黑盒子”

线性表的实现：数组

- Very simple data structure, built into language
- Arrays are very fast at some things
  - Read/write-by-index: O(1)
  - Random-access
- Deletion takes O(N)

数组排序

- 是否需要保持数组被排好序?
- 取决于是否需要执行查找操作
  - Unsorted: search takes O(n)
  - If sorted: search takes O(logn)
- 是否排序会影响到其他操作
  - Unsorted: insert-at-end or delete-at-end takes O(1) (without overflow)
  - Sorted: insert or delete takes O(n)

数组的制约

- 最大的限制：长度固定 数组的长度应该选择多大？
  - If too small, run out of space
  - If too large, wasting space
- Arrays often useful in the following cases
  - Don’t need searches/sorted order
  - Never delete
  - Only delete in reverse-insert order
  - Have fixed amount of data

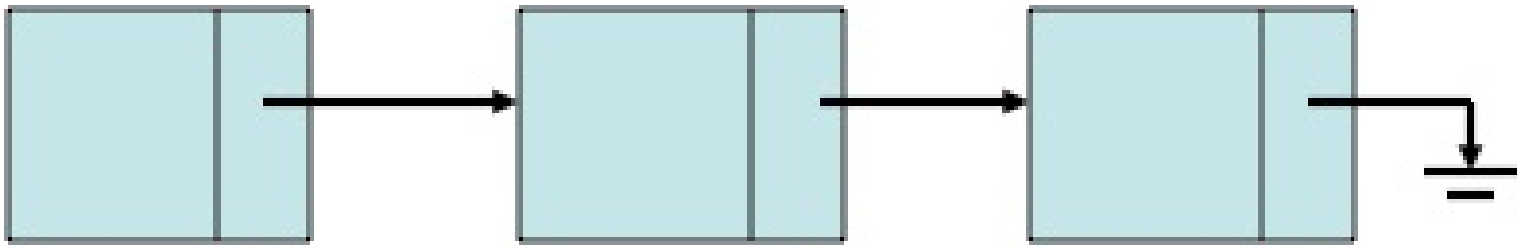
线性表的链表实现

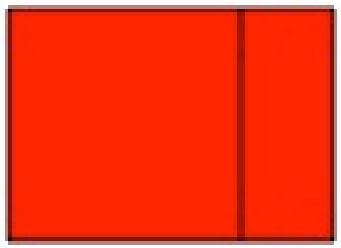
- 数组包含的问题：
  - 空间必须连续
  - 大小必须固定
- New idea: linked list
  - Idea: each element floats in space
  - Contains pointer to “next” + own data
- Remember: don't break the links unless specified

链表操作

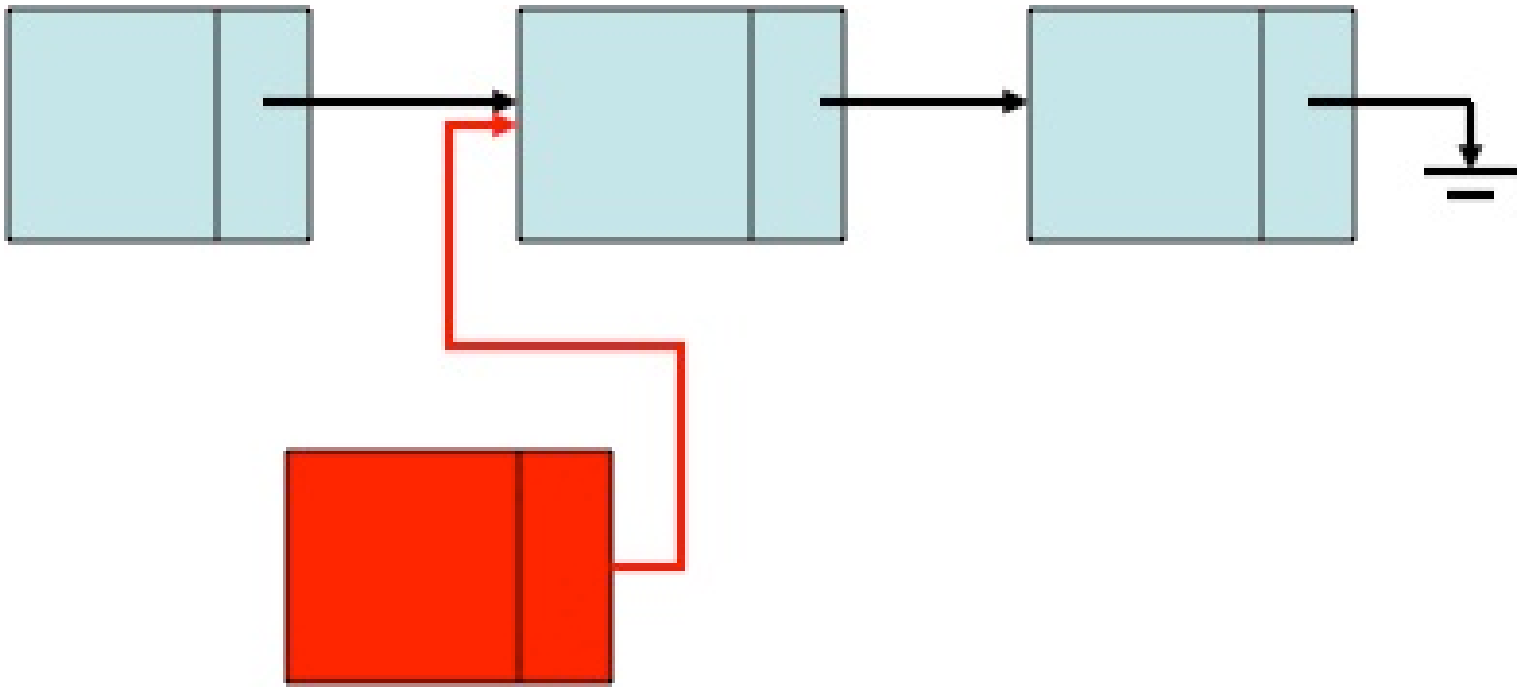
- How to insert-at-pointer (store end pointer): O(1)
- How to delete-at-pointer: O(1)
- Both operations: O(1), no matter what, no sense of "end of list"
  - Never run out of space
  - Downside: coefficient for all may be larger
- Have to obtain new memory each time

链表的操作顺序

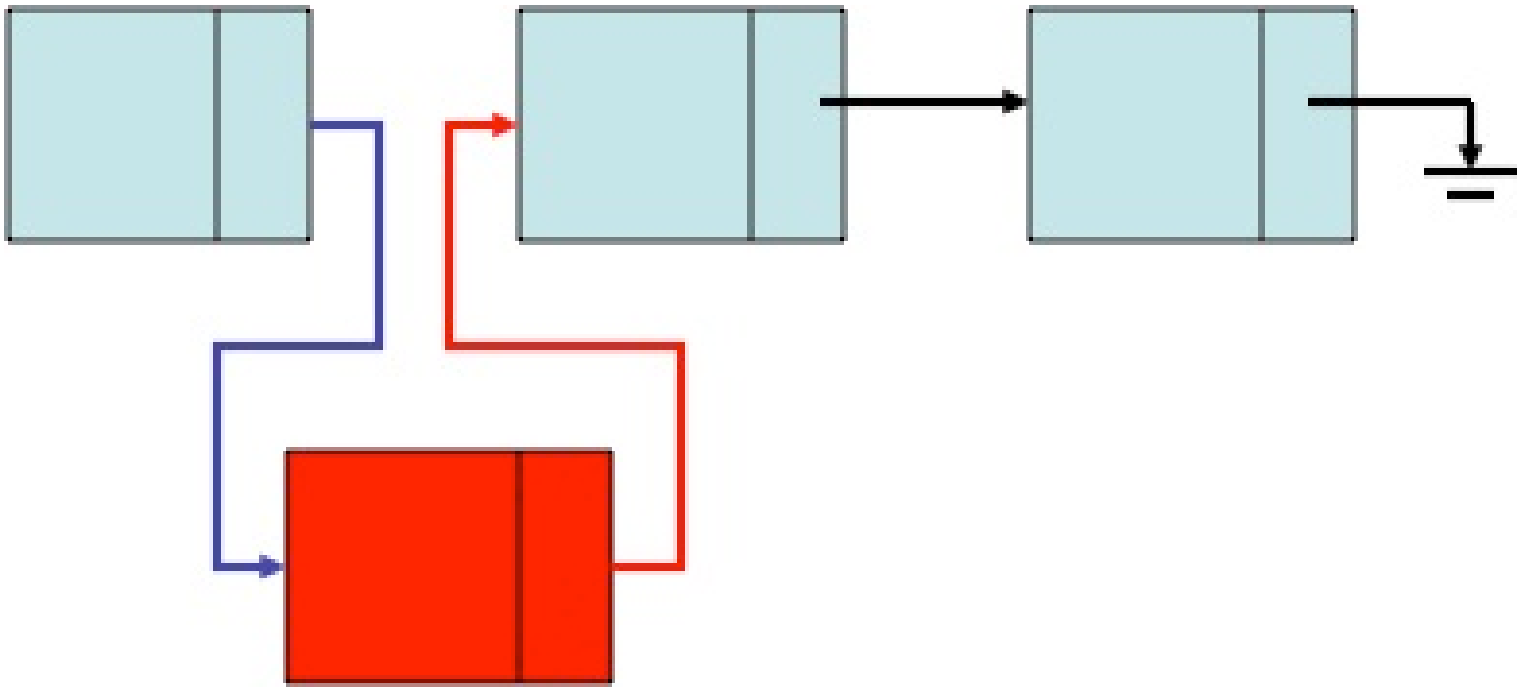




链表的操作顺序



链表的操作顺序



链表代码

```
#ifndef _list_H_
#define _list_H_
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

List MakeEmpty (List L);
Bool IsEmpty (List L);
Bool IsLast (List L);
Position Find (ElementType X, List L);
void Delete (ElementType X, List L);
Position FindPrevious (ElementType X, List L);
void Insert (ElementType X, List L, Position P);
void DeleteList (List L);
Position Header (List L);
Position First (List L);
Postion Advance (List L);
ElementType Retrieve (Position P);

typedef unsigned char Bool;
typedef struct node
{
    ElementType Element;
    Position Next;
} Node;
#endif
```

插入到表头

- List:B $\rightarrow$ C
  - Insert A before B
  - Point A to B
- If B is the pointer to top, then we lose track of A
- Solution: use permanent header pointer
  - H $\rightarrow$ B $\rightarrow$ C
  - Point A to B
  - Insert A: H $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C
- Now insert at top is the same as elsewhere
- Just prevent change of H
- Lesson learned: boundary conditions often cause trouble

指针操作

常见错误：内存访问错误

```
void DeleteList (List L)
{
     Position P;

     P = L->Next;
     L->Next = NULL;
     while (P != NULL)
     {
          free (P);
          P = P->Next;
     }
}
```

双链表

- With regular linked lists, each node knows next node
- Sometimes need to go from node to previous
- Doubly linked list, small change - each node has
  - Data
  - Next pointer
  - Previous pointer

循环链表

- Another option for linked lists: make circular
- Last node points to first (or to header)
- Example: Josephus's problem
- All independent
  - Circular
  - With header
  - Doubly linked

排序链表

- What if sorted:
  - Given pointer, insert/delete is still O(1)
  - Any insert/delete - O(n)
- Usually don't have pointer
- Just have to put in right place
- Can do binary search?
- Skip list: a sorted linked list with additional pointers pointing to farther successors

多重链表

- Suppose each student takes a list of classes
  - Give each student a linked list of selected classes
- Suppose each class has some students
  - Give each class a list of students
- Keep two separate lists of lists?
- Enmesh them in a table
- Compare with 2D arrays

链表和数组的比较

- Why need both?
  - One better for some operations, one for others
- Consider access-by-index: very fast for arrays, O(1) time
- But consider insert
  - Array elements are contiguous, if inserting at end, okay
  - But if in middle, then each after moved back 1, O(n)

Roadmap

用抽象数据结构表示单变量多项式

$$c_n x^n + c_{n-1} x^{n-1} + \ + \ + c_2 x^2 + c_1 x + c_0$$

```
typedef struct polynomial
{
    int degree;
    double coeffs[MAX_DEGREE];
    void (*PolySet) (struct polynomial *P,
                     int degree, double *coeffs)
    void (*insertTerm) (struct polynomial *P,
                        double coeff, int exp) ;
    int (*PolyAdd) (struct polynomial *P1,
                    struct polynomial *P2) ;
    int (*PolyMult) (struct polynomial *P1,
                     struct polynomial *P2);
} Polynomial;
```

多项式相加

```
int PolyAdd (Polynomial *P1, Polynomial *P2)
{
    int i;

    P1->degree = MAX (P1->degree, P2->degree);
    if (P1->degree > MAX_DEGREE)
        return (ERROR);

    for (i = 0; i <= P1->degree; i++)
        P1->coeffs[i] += P2->coeffs[i];

    return (SUCCESS);
}
```

加法举例

$P_1(x) = 10x^5 + 5x^3 + 3x^2 + 8$
$P_2(x) = 8x^5 + 2x$

- Represent both as arrays
- Draw sum as a new array

8	0	3	5	0	10
			+		
0	2	0	0	2	8
			=		
8	0	3	5	2	18

多项式相乘

```
int PolyMult (Polynomial *P1, Polynomial *P2)
{
    int i, j;
    Polynomial P;

    memset (&P, 0, sizeof (P));
    P.degree = P1->degree + P2->degree;
    if (P.degree > MAX_DEGREE)
        return (ERROR);
    for (i = 0; i <= P1->degree; i++)
        for (j = 0; j <= P2->degree; j++)
            P.coeffs[i+j] +=
                P1->coeffs[i] * P2->coeffs[j];
    *P1 = P;
    return (SUCCESS);
}
```

多项式乘法举例

$$P_1(x) = 10x^5 + 5x^3 + 3x^2 + 8$$
$$P_2(x) = 8x^5 + 2x$$
$$P_1 * P_2 = ?$$

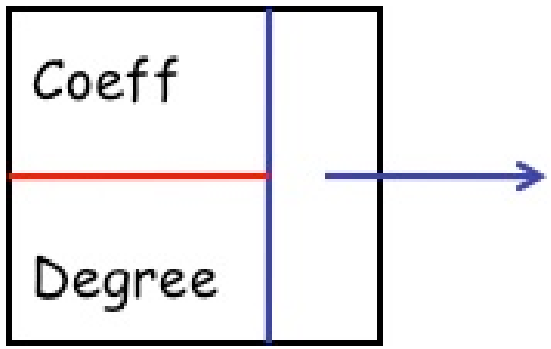
That wasn't so bad, but what about this:

$$P_1(x) = 10x^{500} + 5x^3 + 3x^2 + 8$$
$$P_2(x) = 8x^{500} + 2x$$

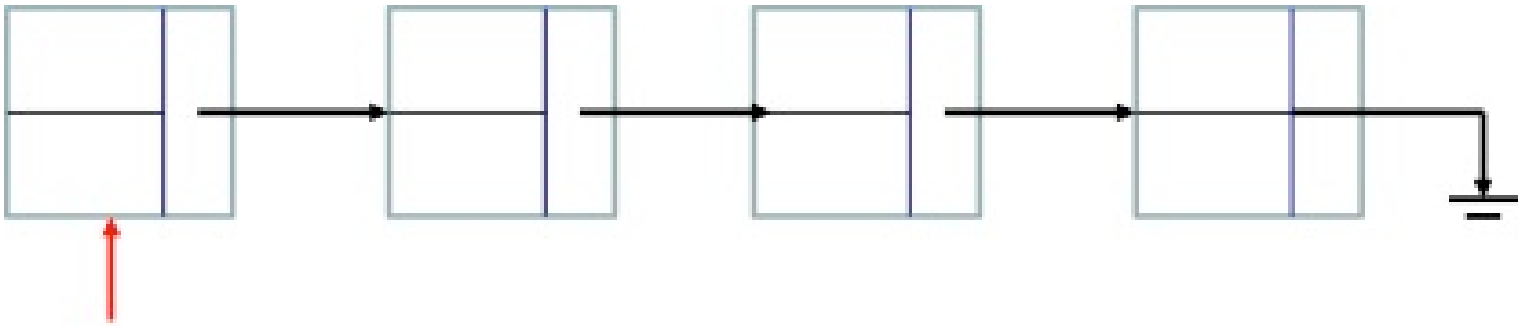
Resulting length: 1,000+1

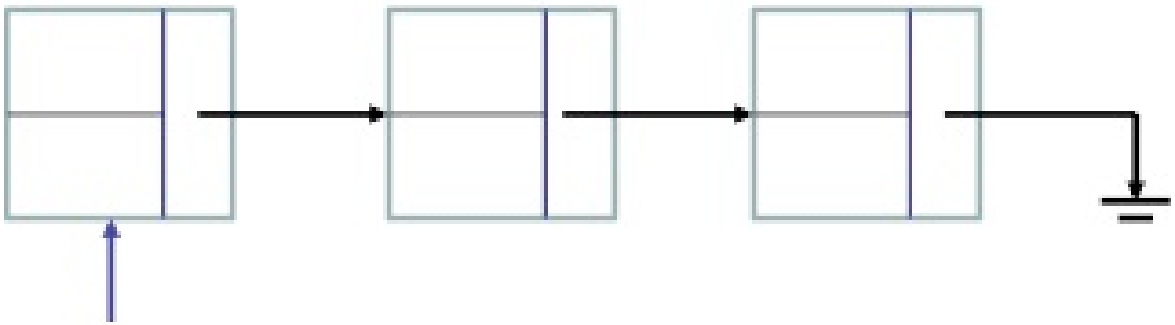
多项式的链表实现

- Replace coefficients array with linked lists
  - Addition gets a little more expensive
  - Multiplication gets much cheaper
- Each node contains
  - Coefficient
  - Pointer-to-next
  - Exponential (no longer implicit)

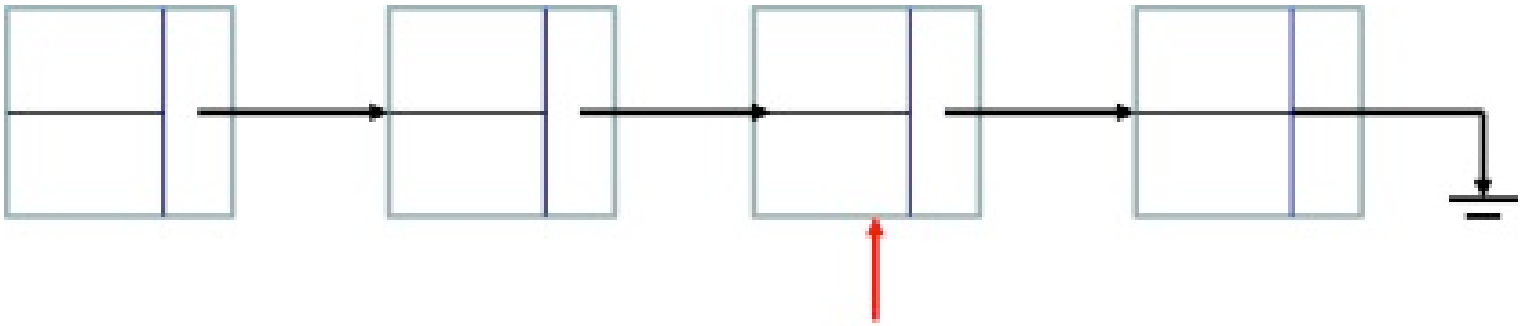


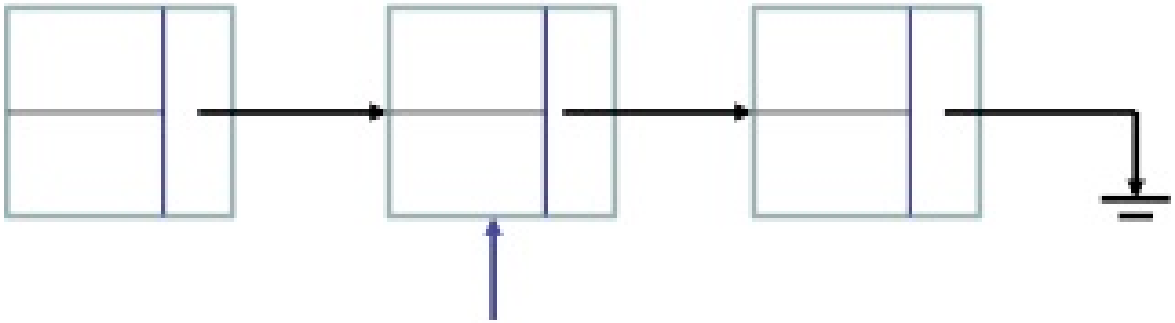
搜索链表



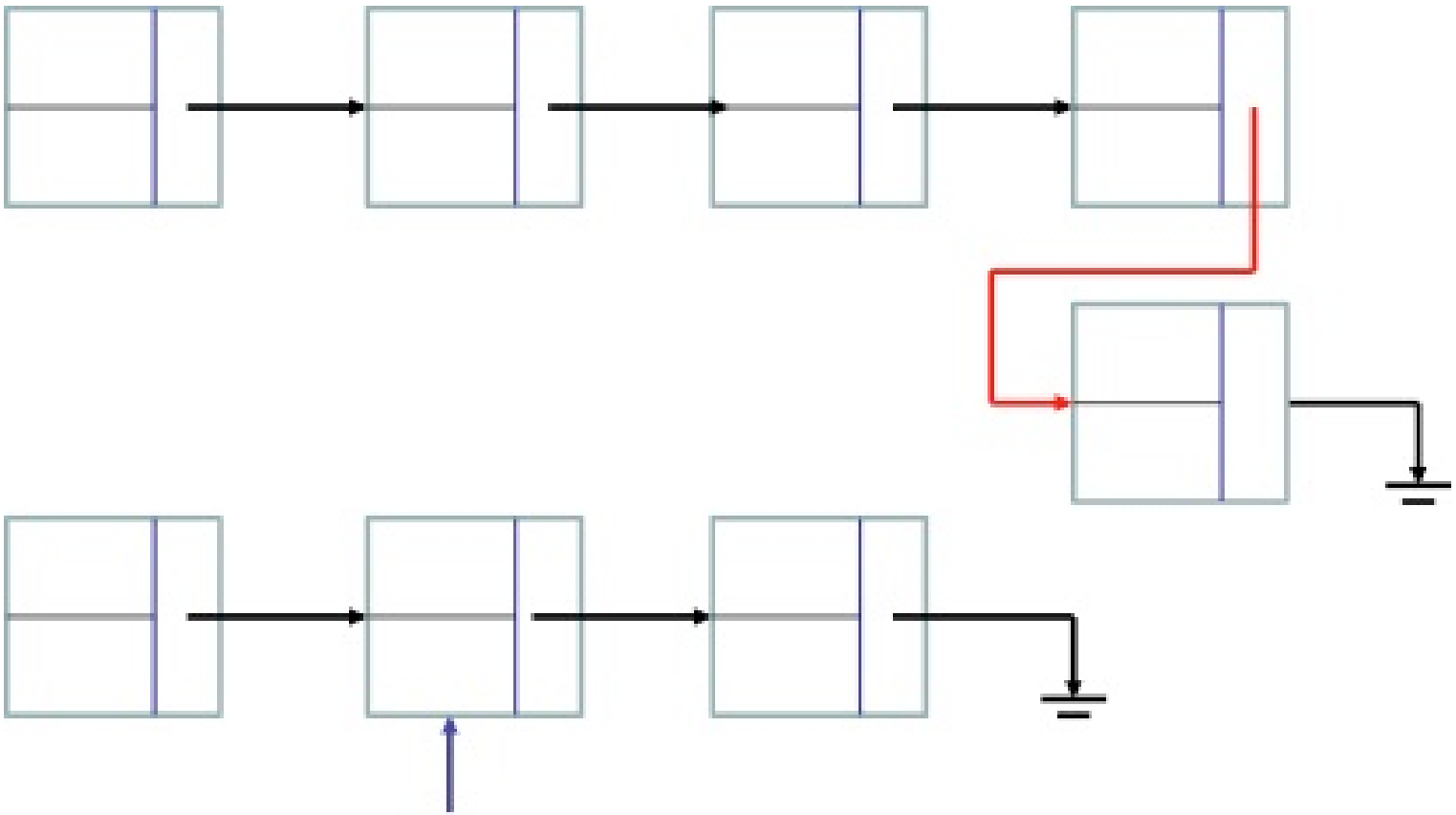


找到同类项





新建结点



链表表示的多项式相加

```
int PolyAdd (Polynomial *P1, Polynomial *P2)
{
    P1->degree = max (P1->degree, P2->degree);
    walk along P1 list and P2 list
    {
        if find same degree
            add coeffiecients and modify P1
        else
            insert P2 to P1
    }

    return (SUCCESS )
}
```

链表表示的多项式相乘

```
int PolyMult (Polynomial *P1, Polynomial *P2)
{
    Polynomial P;

    memset (&P, 0, sizeof (P));
    P.degree = P1->degree + P2->degree;

    walk along P1, P2 lists
    {
        search P for a term, add on or insert
    }
    *P1 = P

    return (SUCCESS);
}
```

实验2：求一个数组的主元

- Pigeonhole Principle
- 对于偶数长度的数组，如果存在主元，必然在数组中存在两个相邻元素同为主元
- 对于奇数长度数组，如果上述情况不成立，若有主元，则最后一个元素必为主元
- Filtering Process:把数组A中A[i]=A[i+1]的元素放入数组B，需要保证A的主元必为B的主元，即淘汰A中的非主元
- 如果不用附加数组，使用A并交换数据
- False Positive/False Negative:需要验证
- Worst case complexity?

主元

- Filtering process: the true majority element must survive, but the surviving element may not be the true majority
- Recursively eliminate candidates until one survives or no one survives
- Must verify

1	2	1	2	1	2	2
1	1	2	2	1	2	
1	2	2	1	1	1	2

递归代码

```
static int majEle_recursive (int a[], int n)
{
    int i, j , k = n/2;
    int tmp;

    if (n == 0) return (NO_MAJ_ELE);

    if (n == 1)  return (a[0]);

    for (i = 0, j = 0; i < k; i++)
    {
        if (a[2*i] == a[2*i + 1])
        {
            tmp = a[j];
            a[j++] = a[2*i];
            a[2*i] = tmp;         /* why do we need this ? */
        }
    }

    tmp = majEle_recursive (a, j);
    if (n % 2 == 1)
    {
        if (tmp == NO_MAJ_ELE)
            return (a[n - 1]);
    }

    return (tmp);
}
```

主元的性质

- Claim: if there is a majority element in the array, and there is no majority element in the beginning from cell 0 to 2*k, the majority element must be the majority of the rest of the array
- Then we can scan the array to see if there is a majority so far; if there isn't up to 2*k, drop the beginning; otherwise, continue checking
- This can be done with a counter, see the candidate majority, increment the counter; see others, decrement this counter; if the counter goes to zero, change the candidate

迭代代码

```
static int majEle_loop (int a[], int n)
{
    int i, count = 0;
    int tmp = NO_MAJ_ELE;

    for (i = 0; i < n; i++)
    {
        if (a[i] == tmp)
            count++;
        else if (count == 0)
        {
            tmp = a[i];
            count++;
        }
        else
             count--;
    }

    if (count == 0)
        return (NO_MAJ_ELE);

    return (tmp);
}
```

实验3

- 3.10，Josephus Problem。用游标方式的循环链表的方式实现Josephus(n，m)问题的求解过程
- 多项式乘法。用链表表示多项式，分别在对指数排序和不排序的情况下，写出求两个给定多项式的乘法的函数。其计算复杂程度分别是多少？
- 按照编程规范编码，实验报告应侧重分析与设计

Roadmap

桶式排序

- 在排序一章将证明在一般模型中，对 n 个对象排序的时间复杂度不会超过nlogn，特殊情况下（增加额外的假设），时间可以更短
- Assume: all numbers < m (max)
- Lay out n buckets: 0..m-1 (playing cards)
- Algorithm
  1. Walk through number array;
  2. Each time see i, put in bucket i
  3. Walk through bucket list, overwrite original list with numbers seen
- Each bucket could be a linked list
- Total time: n + m

基数排序的思想

- Fancier, lower-space bucket sort
- Suppose have written-out numbers in some base B (2 or 10)
- Upper bound on number is given by the max # of digits $D = \log_B m$
- Idea: do bucket sort on each digit instead of all numbers
- Trade time for space
  - Go in which order? Most significant digit first or vice versa
  - One difference with bucket sort: numbers in same bucket are different
  - Must use list/array to store them, not count

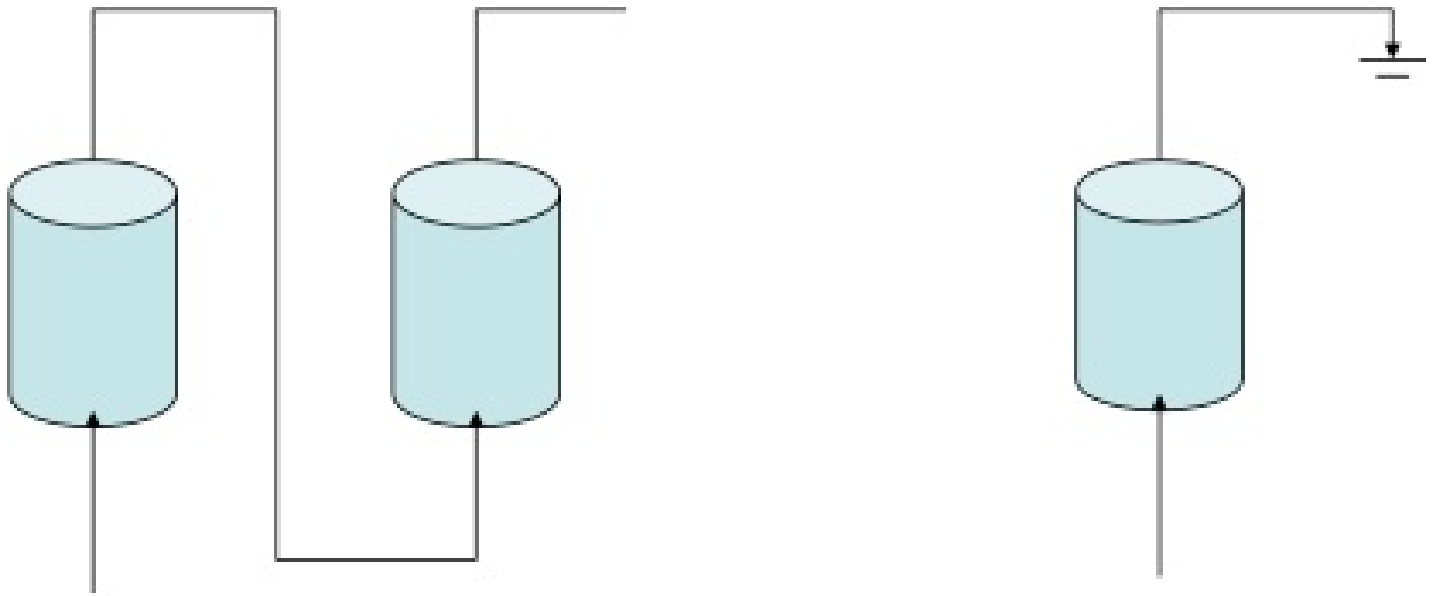
高位优先还是低位优先

- After the first digit gets sorted, write the result back to the original array from the buckets, and go to the next digit to repeat
- Last bucket sort actually puts in order
- Last one has most effect, highest-order digits most important, they're sorted last

```
void RadixSort (int *keys)
{
    while (some key is nonzero)
    {
        bucket sort (keys % 10);
        keys = keys / 10;
    }
}
```

桶的链接

将代表不同数位的桶链接成一个链表，首先把数组倒入桶中，再把桶中的数依次收回数组



对数组使用基于10的基数排序

{8, 20, 59, 30, 21, 40, 10, 11, 22, 34, 23, 5, 33, 65, 44, 75, 17, 66, 77, 88, 99, 9}

排序过程

10									
40					75				9
30	11		33	44	65		77	88	99
20	21	22	23	54	5	66	17	8	59

		23							
9	17	22							
8	11	21	33	44	59	66	77		
5	10	20	30	40	54	65	75	88	99

基数排序的复杂度

- Do regular bucket sort on each digit
- Each rounds: n+10
- D = #digits = log m = #rounds, usually small
- Total: $(n+10)*D = O(n*D) = O(n \log m)$
- Treat D as constant: O(n)
- Can use arbitrary base/#buckets B,
  $O((n+B)D) = O((n+B)\log_B m)$ or O(n+B)

基数排序的正确性

- How come radix works? Because bucket sort is stable, later sorts won't change the ordering of previously sorted digits
- Proof: induction over statement: after step i, numbers are in right order mod b^i
- Base: after step 1, they're right m b^1
- Assume: after step k, they're right m b^k
- Then: run next step; consider some x and y
  - If x[k] > y[k] (or <), then ok
  - If x[k] = y[k] then stable property ensures they stay in same order

选择基数

- $d = \log_B m$: d passes of sort with B buckets
- Complexity $(n + B) \log_B m$
  - Increase B, left up, right down
  - Decrease B, left down, right up
- Suppose n = 1000 = m
  - $B = 2 \rightarrow 1002 * \log_2 1000 \approx 10000$
  - $B = 10 \rightarrow 1010 * \log_{10} 1000 \approx 3000$
  - $B = 50 \rightarrow 1050 * \log_{50} 1000 \approx 1800$
- On the other hand, shift, mod/div by 2^n is faster
- Which matters more, m or n?

基数排序举例

- Can sort all 32-bit integers with 3 passes and 2^{11} buckets
  - Suppose $n = m \approx 4billion$
  - $\log_{2^{11}}(4billion) \leq 3 \rightarrow (2^{32} + 2^{11}) * 3 \approx 12billion$
- Actually, 2^{16} is a much better choice
  - $\log_{2^{16}}(2^{32}) = 2 \rightarrow (2^{3}2 + 2^{1}6) * 2 = 8billion$
  - Bit shift is faster than arbitrary mult/div
  - Remember
    $\log_{10}(4billion) \leq 10 \rightarrow (2^{32} + 10) * 10 \approx 40billion$
- The word RADIX means root in Latin, synonym for base

Roadmap

堆栈

- Definition: ordered sequence whose access is limited to FILO and LIFO
- Simpler than list/list with limitation, but hugely important
- Idea: each item stands on top of the previous
- Each item can be removed is the newest one

户部巷

堆栈操作—链表实现

- Operations allowed: push onto/pop off, think of the cafeteria trays
- Can be implemented with either linked list or array

```
void push (ElementType X, Stack S)
{
    make a new node with value X and pointing to
      S->next and place it to S->next;
}

ElementType pop (Stack S)
{
    advance S->next before returning the
      top element;
}
```

堆栈操作—链表代码

```
#define ERROR    (-1)
#define SUCCESS    (0)
#define NULL    (0)

typedef struct node
{
    int data;
    struct node *next;
} Node;

Node head = {0, (Node *)NULL};

int push (Node *head, int value)
{
    Node *new;

    if (NULL == head)
    {
        printf "(invalid list head\"n);
        return (ERROR);
    }
    new = malloc (sizeof (Node));
    if (NULL == new)
    {
        printf "(malloc () failed, cannot push a new value\"n);
        return (ERROR);
    }
    new->data = value;
    new->next = head->next;
    head->next = new;
    return (SUCCESS);
}
```

堆栈操作—链表代码

```
int pop (Node *head, int *value)
{
    Node *ptr;

    if (NULL == head)
    {
        printf "(invalid list head\"n);
        return (ERROR);
    }
    if (NULL == head->next)
    {
        printf "(empty stack, cannot pop a value\"n);
        return (ERROR);
    }
    *value = head->next->data;
    ptr = head->next;
    head->next = ptr->next;
    free (ptr);
    return (SUCCESS);
}
```

堆栈操作—数组代码

```
ElementType members[MAX];
int top = -1;

Bool isEmpty() {return (top == -1); }
void push (ElementType X)
{
    check if overflow;
    members[++top] = X;
}
ElementType pop ()
{
    check if empty;
    return (members[top--]);;
}
```

堆栈应用

- Implementation pretty simple
- Lots of important applications
- A simple one: balancing symbols
- Expressions in formal languages must be valid
  - HTML/XML
  - Algebra expressions
  - C code
- One simple rule: []s, () can't cross
- Okay: [()], ([]), not: ()(,)()(, [(])

符号平衡

- Algorithm to check

Parentheses Balancing Algorithm

```
step 1.  start with empty stack
step 2.  for each character
           if opening symbol, push
           else if closing symbol, pop
             if empty or not matching
               return ERROR
step 3.  if got through the expression and
the stack is empty
           return SUCCESS
```

- Complexity = #chars

解析中缀表达式

- 10*1.1+5*1.1 = ?
- Normally interpreted as (10*1.1)+(5*1.1) = 16.50
- Windows calculator standard interpretation as ((10*1.1)+5)*1.1 = 17.60
- Without parenthesis, infix notation is ambiguous
- Either way, hard to parse

后缀表达式

- Postfix: operands come before operation
- Compute $a_1 = 10 * 1.1 \rightarrow 10, 11, *$
- Compute $a_2 = 5 * 1.1 \rightarrow 5, 1.1, *$
- Compute $a_1 = a_1 + a_2 \rightarrow a_1, a_2, +$
- Can write 10*1.1+5*1.1 as: 10 1.1 * 5 1.1 * +
- Postfix notation or reverse Polish, Polish (Jan Lukasiewicz) is prefix: + 2 3

解析后缀表达式

- Parsing postfix much easier, no ambiguity
- Simple algorithm

Parsing Postfix Algorithm

walk through expression, for each symbol
if a number, push
else if a binary operation, pop twice, apply operation, push
when done, pop for result

- Example: 5 2 3+8*+3+ 6/ = 8

从中缀到后缀

- No one writes in postfix, so where does it come from?
- Must convert from infix, with the help of a stack
- Idea: using a stack to temporarily keep unprocessed parts
- A bit more difficult than parsing postfix, but not too bad

转换规则

Four rules:

- Read operand $\rightarrow$ print operand
- Read ($\rightarrow$ just push
- Read) $\rightarrow$ pop, print all until corresponding (
  - Parenthesis not printed
  - Don't pop (in any other case
- Read +, * $\rightarrow$ pop stack until operation with strictly lower priority (``+ ''< ``*'' < ``)'') is found, then push
  - Exception: (popped after)

操作顺序与优先级

- a+b*c+(d*e+f)*g → a b c * + d e * f + g * +
- What causes the ambiguity of infix expressions? priority of operations
- Keep lower priority operations in a stack and temporarily won't process it, and reverse the order when a higher priority operation is caught
- Brackets form boundaries of a group of operations, handled separately
- $(a+b)*c+d*(e+f*g) \rightarrow ?$

函数调用

- In programming languages, often talk about heap and stack
  - Heap = dynamic memory
  - Stack = other
- Stack mainly fills up through recursion
- Each function has locals in registers
  - Call another function $\rightarrow$ has own
  - Before overwriting registers, must save backups
  - Write down on activation record (stack frame), push to stack
  - Later, pop and write back
- Function call = (, return =)
- Think of Fibonacci computation by recursion

Roadmap

队列

- Opposite of stack, FIFO/LILO, ``queuing'' in line
- Less important to data structure and algorithm per se
  - Queuing theory
  - Predicting wait times at university cafeteria lines
  - Calls answered in order received
- Servers: print queues, web response queue

队列的实现

- Again two choices: arrays and linked lists
  - Linked lists: obvious, only two operations
  - Arrays: much less so, how to insert? Linear time?
- Idea: save front and rear indices and length number – circular array
- How to test if a queue in array is empty?
  - Use length to keep track of queue usage
  - By the relative position of front/rear?

队列的实现

- Enqueue: A[++rear] = data;
- Dequeue: return A[front++];
- Problems: back goes off left side; front goes off right side
- Make queue circular, front = (front + 1) % MAX

Eg: enqueue 1-5:	[1,f]	[2]	[3]	[4]	[5,r
Dequeue twice	[]	[]	[3,f]	[4]	[5,r
Enqueue 6,7	[6]	[7,r]	[3,f]	[4]	[5]
Dequeue 3times	[6,f]	[7,r]	[]	[]	[]
Dequeue again	[]	[7,r,f]	[]	[]	[]
Enqueue 8	[]	[7,f]	[8,r]	[]	[]
Dequeue twice	[]	[]	[r]	[f]	[]

实验3

- 游标：对应于指针，内存可视为一个字节数组，地址即内存数组的索引
- 游标表示链表：
  - 数组分配完毕，需要确定哪些成员可以分配，哪些已被使用：用一个单链表连接所有未被分配的成员
  - 需要替代malloc()和free()：将上述链表的第一个节点分配出去，把回收的节点插入到上述链表的头结点后

小结

- 抽象数据结构：数据＋操作
- 线性表既可以用数组实现，也可以用链表实现，各自有自己的优缺点
- 链表操作中要注意指针的正确使用
- 堆栈和队列所定义的插入和删除操作需要遵守特定的顺序
- 堆栈具有广泛的应用，尤其是需要完成倒序时可以考虑使用堆栈

实验4

- Radix Sort。实现桶式排序和基于桶式排序的基数排序。在基数B，数组长度n和最大元素值m中，对排序时间影响最大的是哪一个？元素在未排序数组中的顺序是否对时间复杂度有影响？设计试验证明你的想法
- Stack。用C语言设计堆栈，并实现中缀表达式到后缀表达式的转换