

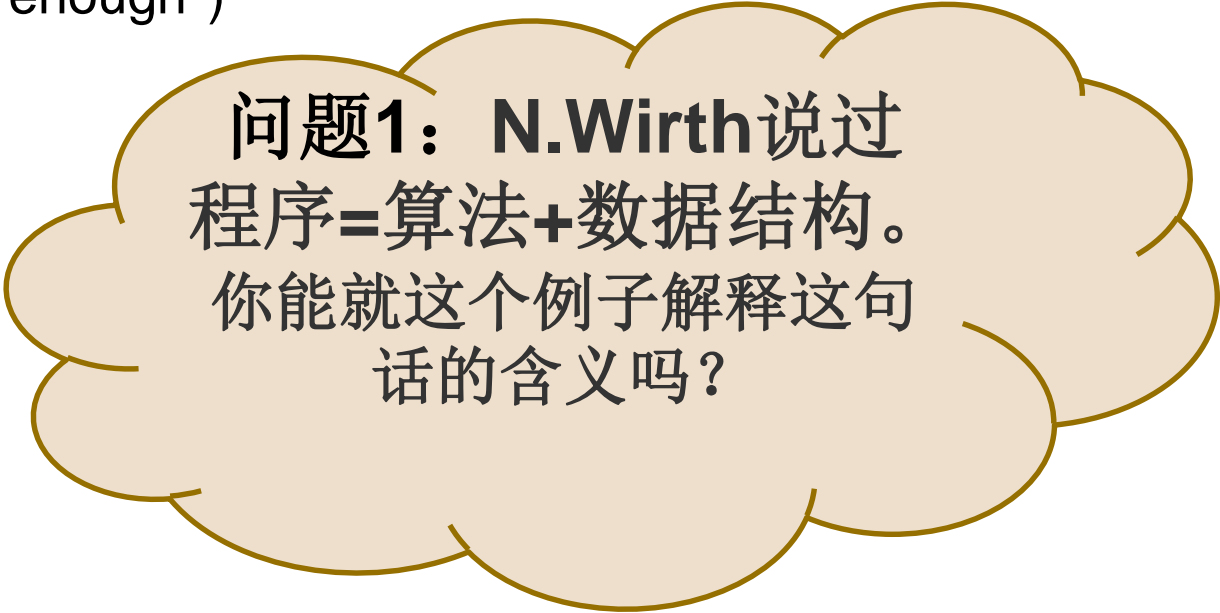
计算机问题求解 — 论题2-11

- 基本的数据结构

2016年04月28日

问题：任意输入一行字符串，以#结束。输出这个字符串的反串

```
Program reverse(input, output);  
  var elements: array[0:n] of char;    (*n is big enough*)  
      index: int;  length: int; c:  char;  
begin  
  read(c); length:=0;  
  while (c<>'#') do  
    begin  
      elements[length]:=c;  
      length := length+1;  
      read(c);  
    end  
    for index:=length-1 to 0 do write(elements[index]);  
  end.
```



**问题1：N.Wirth说过
程序=算法+数据结构。
你能就这个例子解释这句话的含义吗？**

问题2:

你能从上例中的” `var elements: array[0:n] of char;` ”以及某个具体输入 “`damrnqiz`” 中解释以下几个概念的关系吗？

数据；数据类型；数据结构

我们引入Dynamic set这个概念的用意是什么？

问题3：你除了能看出动态集合上常见的操作外，能否看出“结构”来？

SEARCH(S, k)

A query that, given a set S and a key value k , returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S .

INSERT(S, x)

A modifying operation that augments S with an element pointed to by x . We usually assume that x is not already in S . We usually assume that x is not already in S .

DELETE(S, x)

本质上，我们所采用的所有表达动态集合的高级数据结构，定义其上的操作，少不了上述基本功能

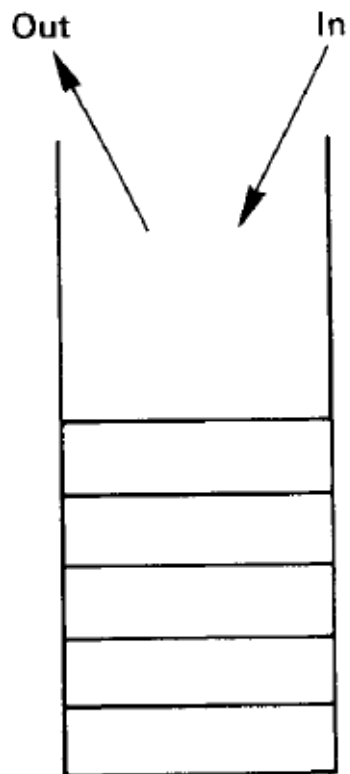
SUCCESSOR(S, x)

A query that, given an element x whose key is from a totally ordered set S , returns a pointer to the next larger element in S , or NIL if x is the maximum element.

PREDECESSOR(S, x)

A query that, given an element x whose key is from a totally ordered set S , returns a pointer to the next smaller element in S , or NIL if x is the minimum element.

问题4：我们只能采用数组来组织和管理动态集合吗？



Roughly, a stack is a set of objects which arrive at different points in time and are added to the set. When elements are deleted from the set, the object which was inserted last is removed first.

A stack can be visualised as a pile of objects where objects can be added to or removed from the pile only from the top (see figure 2.3).

Figure 2.3 Pictorial model of a stack.

其实，动态集合的不同组织方式，可能带来算法的变化和效率的变化

- 定义栈的管理动作：
initialize(S):创建一个栈;
push(S,c):将c压入栈S中;
top(S):栈顶元素;
pop(S):将栈顶元素去除;
empty(S):判定S是否为空
full(s):判断S是否满

```
Program reverse(input, output)
var s: stack; c: char
begin
  initialize(s);
  read (c);
  while c<>"#" do
    begin
      push(s,c);read(c);
    end
  while not empty(s) do
    begin
      write(top(s));
      pop(s);
    end
  end
end
```

但是，上述程序是无法执行的：

- 大多数语言并没有像提供array, record, pointer那样提供stack这样的语言设施供我们使用：
 - 如此的数据抽象，不是都被语言支持
- 我们还会根据应用需求、管理需求进行这样那样的抽象，构造自己的数据管理方式：
 - 队列、链表、树、图.....

问题5：至此，你必须能清楚的理解什么叫“定义并实现一个数据结构”

数据结构stack的定义

stack: *set* of elements of type *elemtype*

(* *elemtype* is used to refer to the type of the individual elements in a stack.
Elemtype can potentially be any defined type *)

Operations:

procedure initialise(var S : stack);

This procedure assigns an empty stack to S.

procedure push(var S : stack; a : elemtype);

Stack S should not be full. This procedure adds the element a to the top of the stack.

procedure pop(var S : stack);

Stack S should not be empty. The top element is removed.

function top(S : stack) : elemtype;

Stack S should be non-empty and this function returns the top element of the stack S. The stack S is left unchanged.
(If *elemtype* is a structured type, this function should be rewritten as a procedure, see note 1 of section 3.1.)

function empty(S : stack) : boolean;

This function returns true if S is empty and false otherwise.

function full(S : stack) : boolean;

If S is full this function returns a true value.
Otherwise it returns a false value.

问题6：定义一个数据结构，必须定义哪些东西？

用高级语言提供的基本数据类型和数据结构来实现在自定义结构中定义的数据和操作

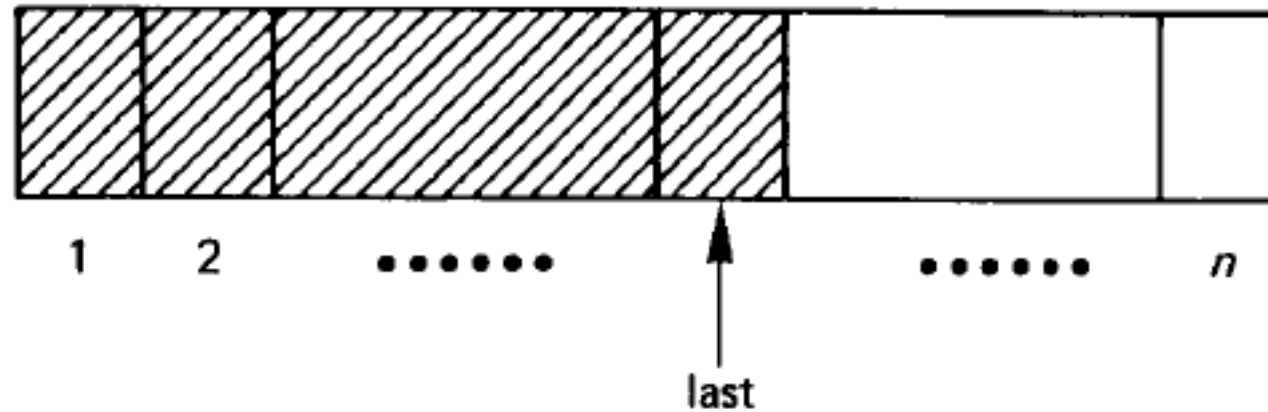


Figure 2.6 Array implementation of stacks.

数据实现部分:

```
const n =    ;           (* n is the maximum size of a stack *)
type stack = record
    elements : array [1..n] of elemtype;
    last : 0..n      (* 0 signifies an empty stack *)
end;
```

操作实现部分

```
procedure initialise (var S: stack);  
  var i : integer;  
  begin  
    for i := 1 to N do S.elements[i] := elem0;  
    (* elem0 is an element of the type of S *)  
  end;
```

一旦我们小心翼翼地完成了定义和实现，我们就可以发布这个“**数据结构**”为一个“**数据类型**”供别的程序员直接使用，他们不用关心这个类型的所有实现细节，就像我们自己使用**int**类型一样！

```
function top (S: stack) return elem;  
  begin  
    (* return the top element of the stack *)  
    top := S.elements[S.last]  
  end;
```

实际上，我们还可以采用不同的实现方式来实现某个数据管理方式！

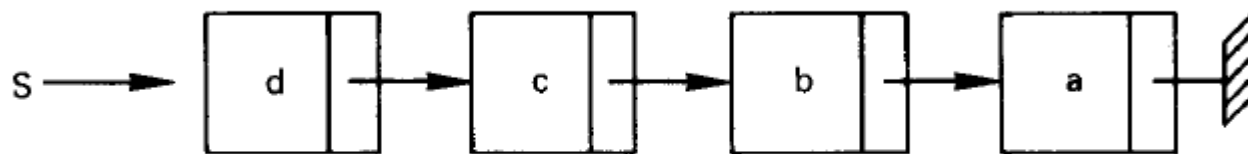


Figure 3.2 Pointer representation of a stack S.

- 难道我们就定义了两个不同的**stack**了吗？

如果我们在构造自己的数据组织方式时，写出
了一个关于这个方式的“约定”，而暂时没有
涉及实现细节，甚至不再关心实现细节而交由
其他人员实现时，我们写出来的“约定”就有
了新名词：

抽象数据类型：ADT

完整的 stack的 ADT

ADT stack: *set* of elements of type *elemtype*

(* *elemtype* is used to refer to the type of the individual elements in a stack.
Elemtype can potentially be any defined type *)

Operations:

procedure initialise(**var** S : stack);

This procedure assigns an empty stack to S.

procedure push(**var** S : stack; a : elemtype);

Stack S should not be full. This procedure adds the element a at the top of the stack.

procedure pop(**var** S : stack);

Stack S should not be empty. The top element of the stack is removed.

function top(S : stack) : elemtype;

Stack S should be non-empty and this function returns the top element of the stack S. The stack S is left unchanged. (If elemtype is a structured type, this function should be rewritten as a procedure, see note 1 of section 3.1.)

function empty(S : stack) : boolean;

This function returns true if S is empty and false otherwise.

function full(S : stack) : boolean;

If S is full this function returns a true value.

Otherwise it returns a false value.

以定义ADT为目标去定义数据结构的好处：

- 关注分离
- 信息隐藏
- 模块化设计
- 可以加载形式化研究

相当重要的“加载形式化研究”！

Finally, since ADTs are mathematical objects, their meaning can be formally specified. Based on these specifications, it is then possible to verify that a given implementation of an ADT is correct and agrees with the specification.

狭义的程序设计中难得一见的科学内涵！

Stack的数据部分的形式约束

```

stack : : c      : integer      (* current time-point *)
      max      : integer      (* maximum allowable size *)
      elems    : pair-set     (* elems is a set of pairs *)
pair   : : object : elemtype    (* object is of type elemtype *)
      t        : integer      (* t is a time-stamp *)

```

invariance 1: not $[\exists a, b, t \ (a, t) \in \text{elems and}$
 $(b, t) \in \text{elems and}$
 $a \neq b]$

invariance 2: $\forall (a, t) \in \text{elems } c > t$

invariance 3: $|elems| \leq \max$

Stack 摺

the previous time-stamp c' and the pair (a, c) belong to $S *$)

pop(s)

pre : $|elems| > 0$

(* s is not empty *)

post : $\exists (a, b) \in elems$ such that

(* (a, b) is the element in s with largest

$[\forall (p, q) \in elems \ b \geq q]$

time-stamp b and is deleted from

and $elems' = elems - \{(a, b)\}$

elems *)

initialise(s)

pre

top(s)

post

pre : $|elems| > 0$

post : $\exists (a, b) \in elems$ such that

(* (a, b) is the element in S with

$[\forall (p, q) \in elems \ b \geq q]$

largest time-stamp b and a is

and $top(s) = a$

returned as the value of the

function. Note that c and s are

not changed *)

push(s, a)

pre

post

than

empty(s)

pre : true

post : $empty(s) = (|elems| = 0)$

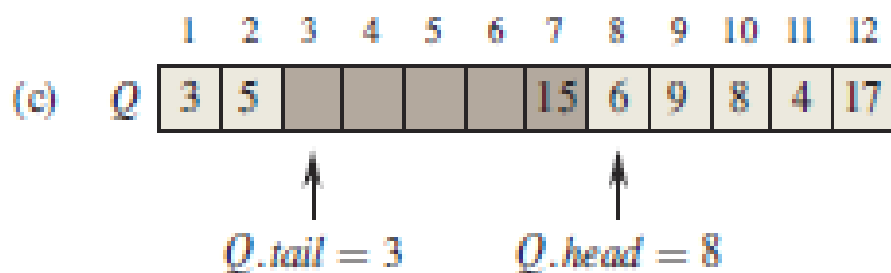
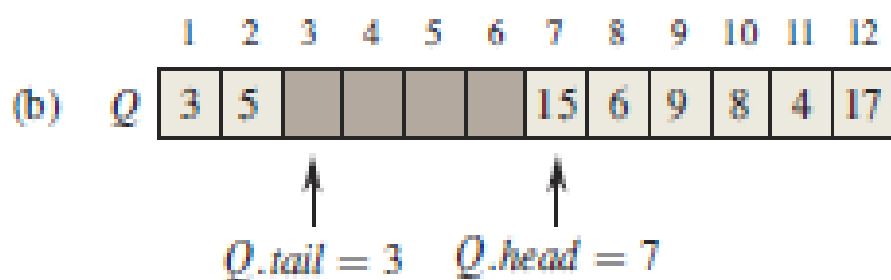
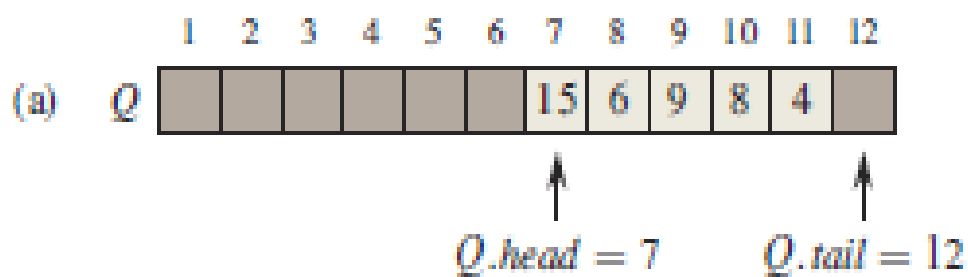
full(s)

pre : true

post : $full(s) = (|elems| = max)$

问题7:

你认为队列与栈与其元素的物理位置分布有关吗?



问题8:

你能举出一些例子，说明queue这种结构中哪些数据成分必须有？它们必须满足什么性质？哪些操作必须提供，它们又必须遵循什么约定？

问题9:

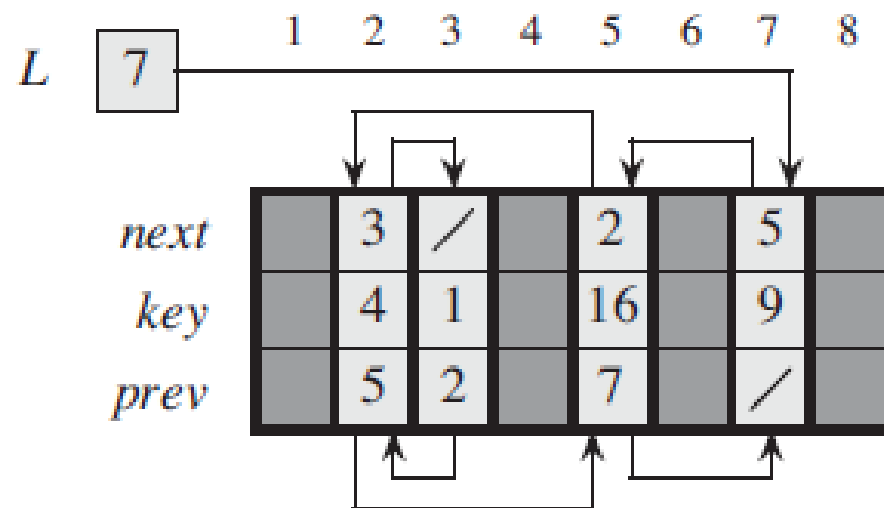
栈与队列的本质差别是什么？
为什么我们需要这种差别？

问题10:

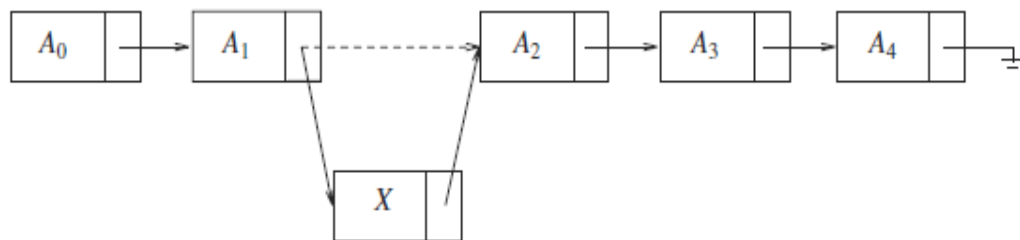
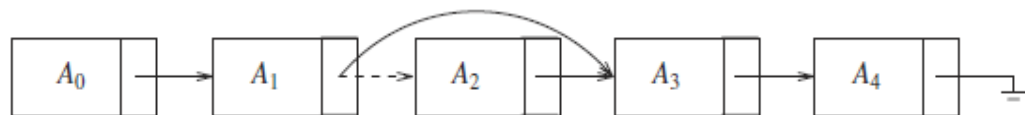
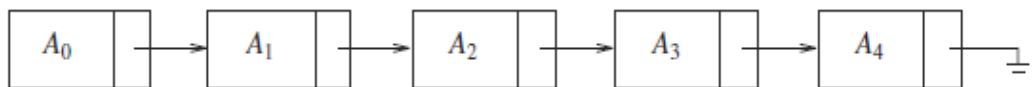
你觉得在我们的讨论的“结构”的背后是否有什么基本的数学概念吗？

问题11:

你能就下两例说说看，指针到底是什么？

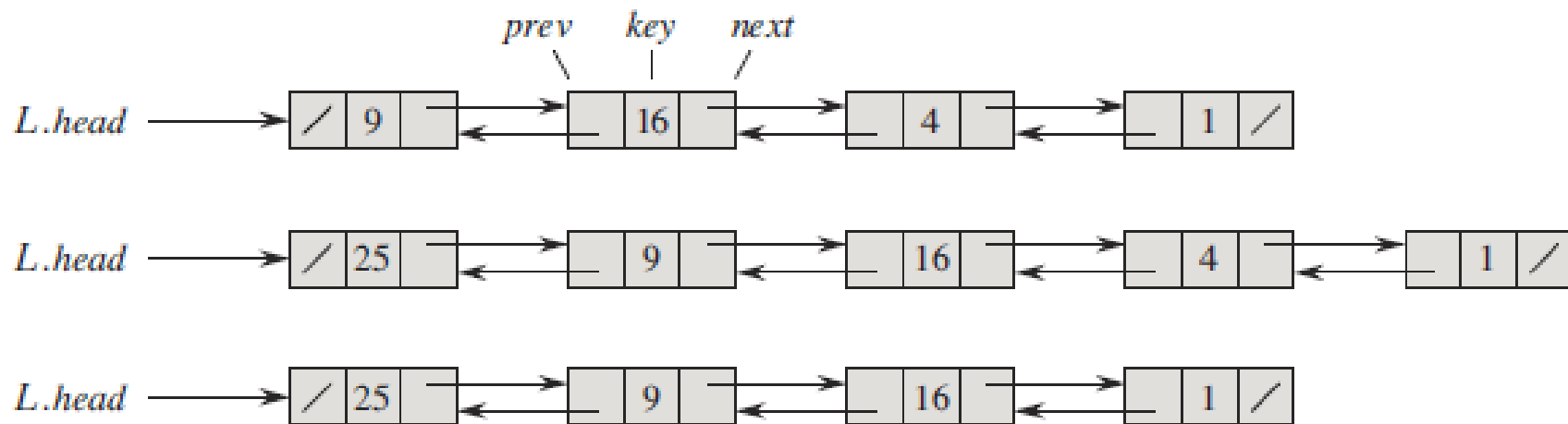


链表：一切操作均围绕指针



问题12:

这个结构有什么不便之处吗？



LIST-INSERT (L, x)

- 1 $x.next = L.head$
- 2 if $L.head \neq \text{NIL}$
- 3 $L.head.prev = x$
- 4 $L.head = x$
- 5 $x.prev = \text{NIL}$

LIST-DELETE (L, x)

- 1 if $x.prev \neq \text{NIL}$
- 2 $x.prev.next = x.next$
- 3 else $L.head = x.next$
- 4 if $x.next \neq \text{NIL}$
- 5 $x.next.prev = x.prev$

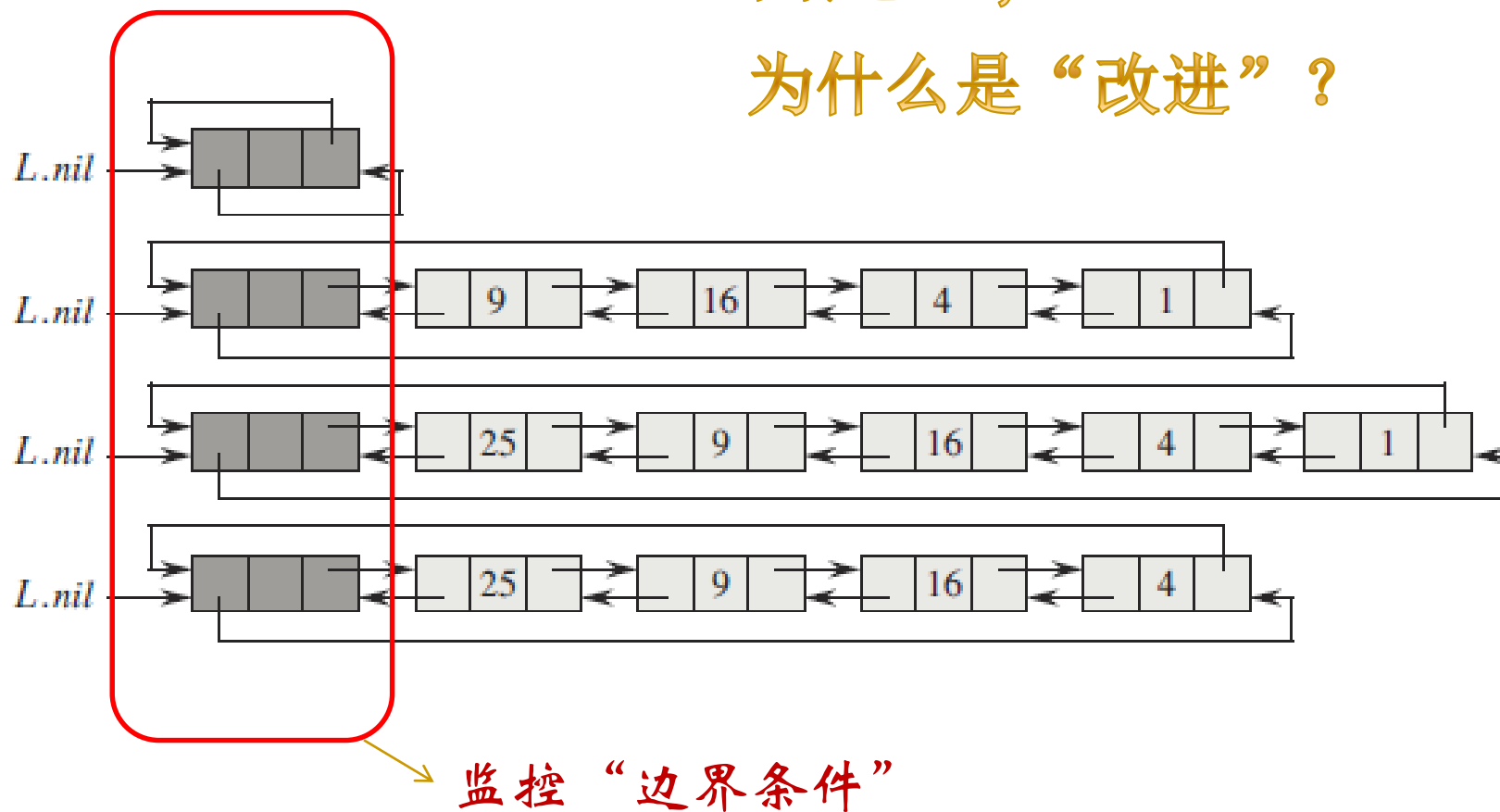
问题13:

删除一个对象的代价是多少？

进一步的改进

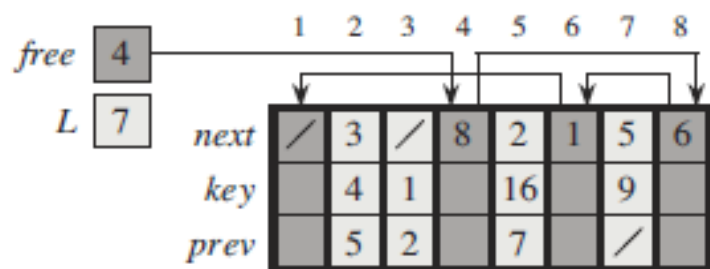
问题14;

为什么是“改进”？

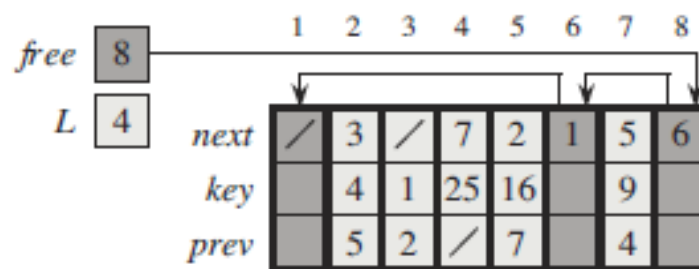


两列表单数组实现

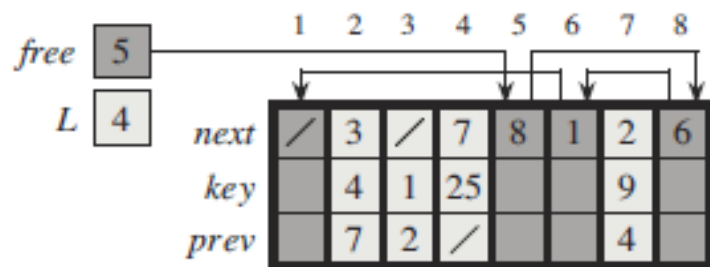
- 基本原理：在一个数组空间维护两个链表，“实际数据”和“可用空间”。“此消彼长”



(a)



(b)

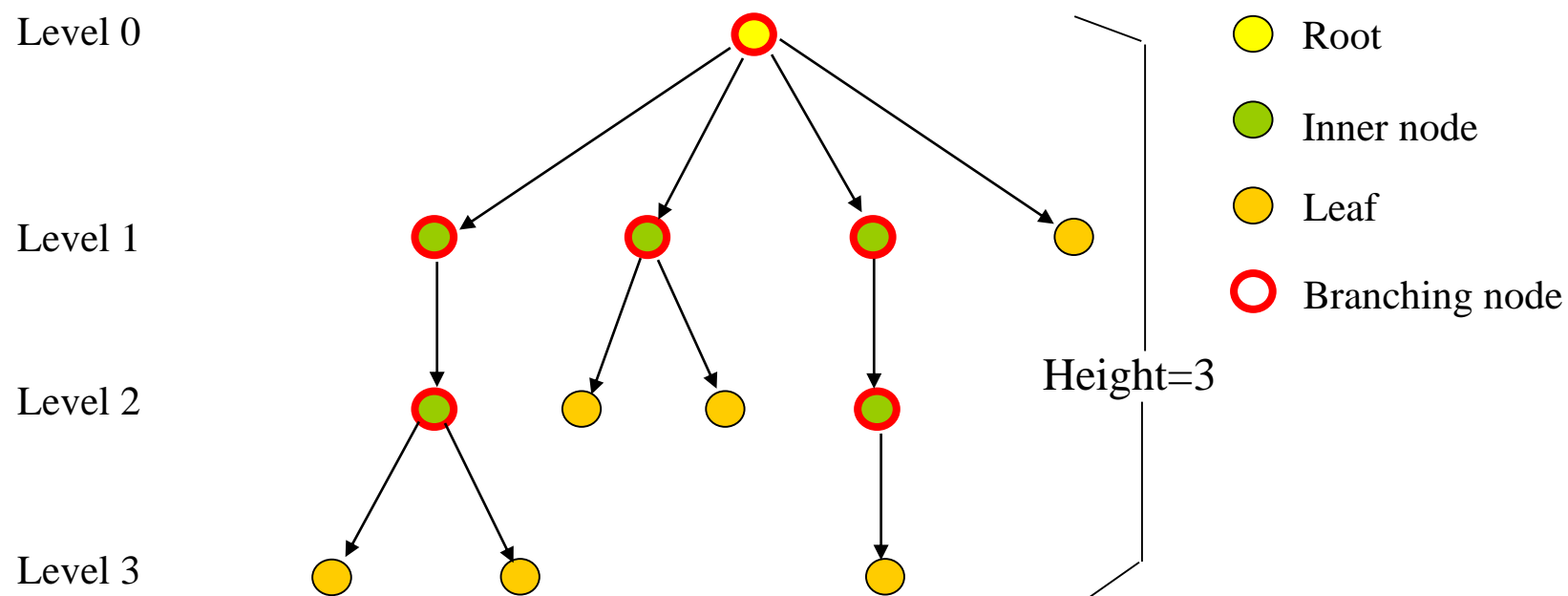


(c)

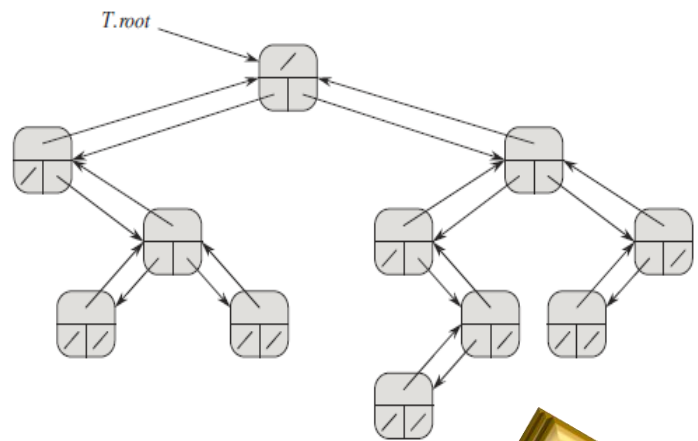
问题15:

你能解释一下
这几个图吗?

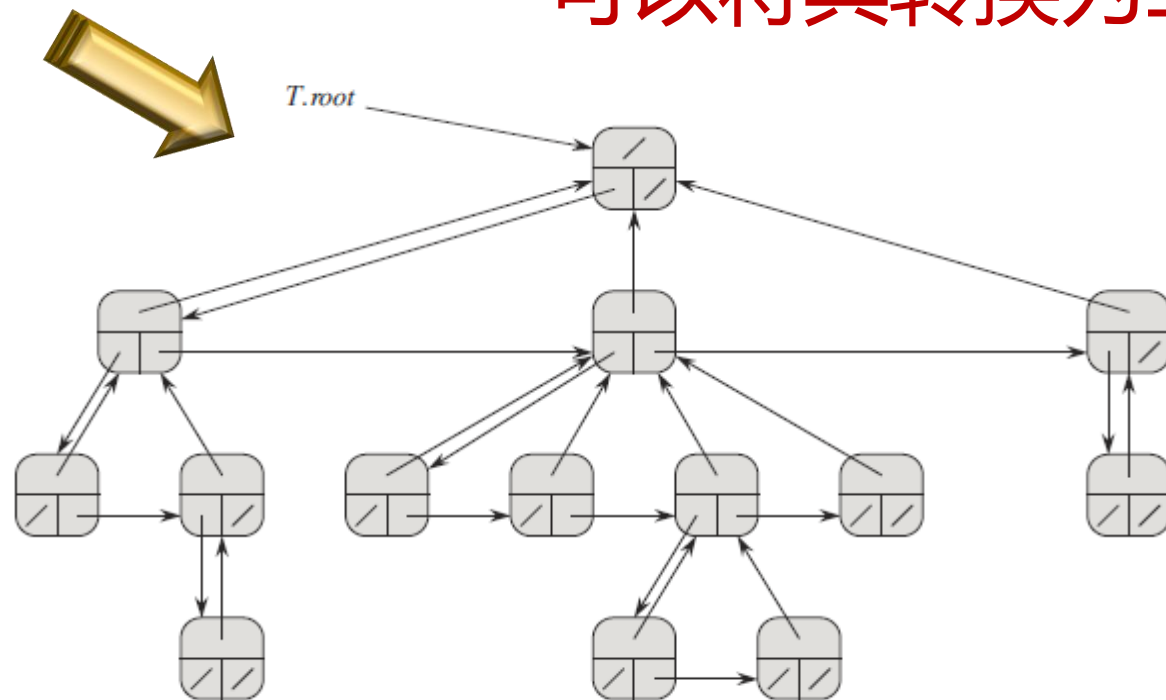
根树: 如何从链表的角度看它?



如果任意结点最多有两个子结点, 则该根树成为二叉树(binary tree), 显然用指针实现链表的方法很容易扩展到二叉树



这其实意味着：
即使问题逻辑需要多叉树，我们也可以将其转换为二叉树来实现。



课外作业

- TC pp.235-: ex.10.1-4; 10.1-5; 10.1-6
- TC pp.240-: ex.10.2-1; 10.2-2; 10.2-3; 10.2-6;
- TC pp.245-: ex.10.3-4; 10.3-5;
- TC pp.248-: ex.10.4-2; 10.4-3; 10.4-4
- TC pp.249-: prob.10-3