Ch6. Data types

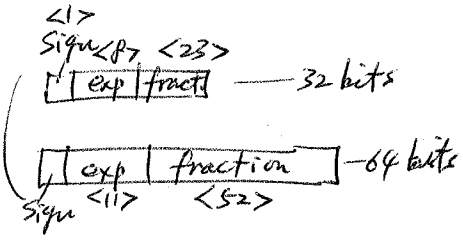
— a data type — defines a collection of data values and a set of predefined operations on those values.

- user defined abstract data types — ch11.
- structured (non-scalar) data types
  array, record/struct
- object — user-defined and language-defined abstract data types — ch. 11, 12.

§6.12. primitive data types

— numeric types

- int
- float — single/double precision
- complex — python: (7+3j) or (7+3J)
- decimal



Boolean — C/C++ Boolean types allows numeric expressions.
Java/C# — no.

Char. type

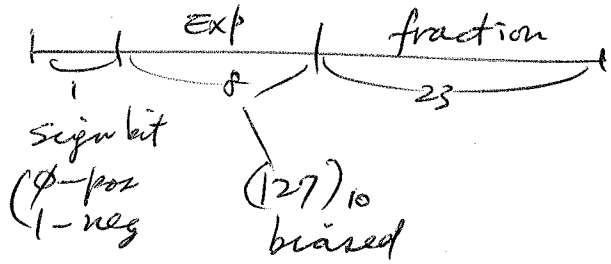
ASCII — 8 bits — (0 ~ 127)

Unicode — 1991 (UCS-2 standard — 16 bits)

- first 128 ≡ ASCII
- Java uses first
- Javascript, python, perl, C#, F#

UCS-4 (UTF-32) — 32 bits char. set.
2000

floating point type



— single precision (fp.)
(32 bits)

ex) $78.375)_{10} = 1001110.011)_2 \Rightarrow$ (scientific format: $1.001110011 * 2^6$
(6 positions move)

$78)_{10} = 1001110$

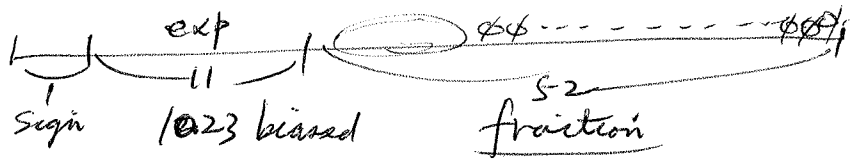
$0.375)_{10} = 3/8 = \underbrace{1/4}_{2^{-2}} + \underbrace{1/8}_{2^{-3}} = 0.011$

— add bias $127)_{10}$ to the exponent ($E = 6$)

$\Rightarrow 127 + 6 = 133$

$$\underbrace{0}_{\text{sign } (+)}\ \underbrace{10000101}_{\text{exp (8 bits) } 133)_{10}}\ \underbrace{0011\,1001\,1000\,0000\,0000\,000}_{\text{fraction (23 bits)}}$$

— double precision (64 bits)



§6.3. Character string types

— values consist of sequences of char's

ex) char str[ ] = "apples"; → | a | p | p | l | e | s | ∅ |

array of chars. null char.

operation: strcpy, strcat, strcmp, strlen

C-string library — insecure — ex) strcpy (dest, source) — mem. overwriting

|20| |50|

⇒ string class in C++
<string>

Java — string class — const. strings
string Buffer class — changeable strings, arrays of single chars

Similar in C#, Ruby

python — string is a primitive type

— pattern matching capabilities — using regular expression

C++, Java, python, C#, F# — in class libraries

ex) /[A-Z a-z][A-Z a-z \d]+/ — strings that begin with a letter followed by one or more letters or digits.

all letters — all letters and digits

— string length options

- static length strings — python, C++, Java, Ruby, C#, F#
- limited dynamic length strings — ∃ max. bound — C, C++
- dynamic length strings — Javascript, perl, C++

no special dynamic storage alloc.

3 possible storage management schemes.

1. linked-list way storage in heap; — disadv.: pointer chasing, links — space needed
2. arrays of pointers to individual chars. in the heap; — processing is faster than (#1).
3. Complete string as adjacent cells in the heap;
   → less storage, but alloc/dealloc — slower.

book note

- Enumeration types — define group(collection) of <u>named constants</u>.

  ex) type day = (Mon, Tue, Wed, ...)

  ⇒ Mon < Tue < Wed ...

- short-circuit evaluation of Boolean expression

  ex) C. if (⬭ || ▨)
  if (⬭ && ▨)

  — 2nd condition expression is evaluated <u>only if necessary</u>

- type conversion
  - coercion — automatic conversion between types
  - type casting — explicit

  ex) C. char's are implicitly coerced to integers

```
int c;
c = getchar();
while (c != EOF)
{ putchar(c);
  c = getchar();
}
```

§.6.5 Array types

binding to subscript ranges and storage

- static array
  - statically bound
  - storage allocation is static (before run time)
  - Adv: efficiency
  - disadv: storage is fixed
  - ex) C/C++ static array in a func.
- fixed stack-dynamic array
  - statically bound
  - storage alloc. is done at declaration elaboration time during execution (at run time stack)
  - Adv: space efficiency
  - disadv: alloc/dealloc. time overheads
  - ex) C/C++ non-static array in a func.
- fixed heap-dynamic array
  - subscript range binding
  - storage binding in heap
  - when user program requests during execution.
  - → once created, keeps same range and storage.
  - Adv: flexibility
  - disadv: alloc. time in heap is longer than in stack.
  - ex) C — malloc/free
    C++ — new/delete
    Java — all non-generic arrays
- heap-dynamic array
  - subscript range binding
  - storage alloc. in heap
  - dynamic and can change any number of times during exec. (array's life time)
  - Adv: flexibility
  - disadv: alloc/dealloc. overheads
  - ex) C# List class
    - List<string> stringList = new List<string>();
    - stringList.Add("xxx");
    - ...

— array operations

- assignment
- Catenation
- Comparison (equality)
- slices
- ...

slices — substructures of an array as a unit

⇒ python

```
[0] [1] [2] ...
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [ [1,2,3], [4,5,6], [7,8,9] ]
        [0]      [1]      [2]

vector[3:6] = [8, 10, 12]
```

1st index — not included

[3, 6)

```
mat[1] = [4, 5, 6]
mat[0][0:2] = [1, 2]
```

— in mat[0], index [0, 2) included, not included

```
vector[0:7:2] = [2, 6, 10, 14]
```

[0, 7) every 2

⇒ C++ Substring

```
string s1 = "....";
string s2 = s1.substr(0, 5);
```

start index, length

⇒ s1 = "1234.567"

s2 = s1.substr(0, s1.find(".")) ;

⇒ s2 = "1234"

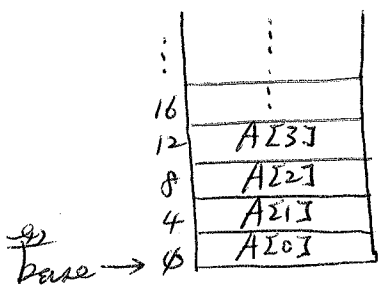
— Implementation of Array types

<u>1-D array</u> — a list of adjacent memory cells.

— <u>1-D array layout</u>

with lower bound ϕ (C/C++,..)

eg) int A[10];



— formula for computing <u>address of A[i]</u>

$$\boxed{i * w + base}$$

w: size of element (in byte)

eg) @A[3] = (3 * 4) + ϕ = 12 byte addr.

— <u>2-D array layout</u>

with lower bound ϕ (C/C++,..)

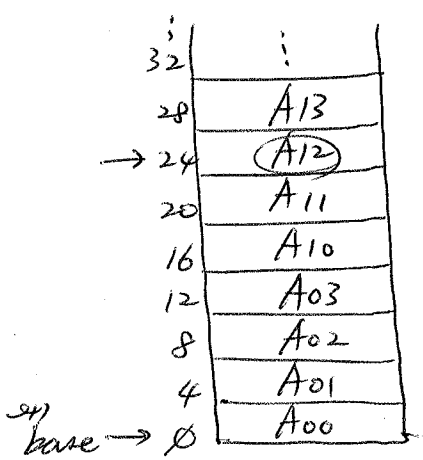
eg) int A[H_1][H_2];

H_1: # of rows, H_2: # of cols

<u>int A[3][4];</u>

A_{00}	A_{01}	A_{02}	A_{03}
A_{10}	A_{11}	A_{12}	A_{13}
A_{20}	A_{21}	A_{22}	A_{23}

1. <u>row-major layout</u>



— formula for computing <u>address of A[i1][i2]</u>

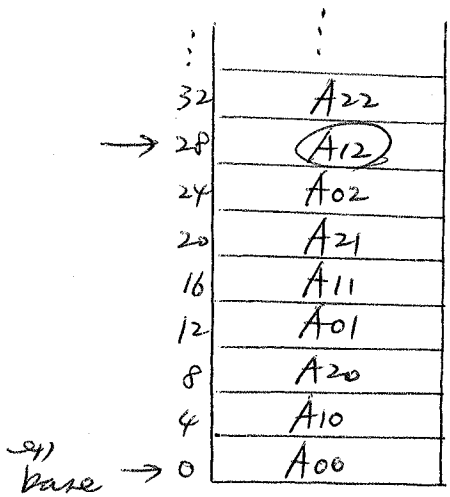
$$\boxed{[(i1 * H_2) + i2] * w + base}$$

H_2: # of cols, w: size of ele. (in byte)

eg) @A[1][2] = ((1*4)+2)*4 + ϕ (base addr.)

= 24 (byte addr.)

2. col-major layout



— formula for computing address of A[i1][i2]

$$\boxed{[i1 + (i2 * H1)] * W + base}$$

H1: #of rows; W: ele. size (byte)

eg) @ A[1][2] = (1 + (2 * 3)) * 4 + ∅ (base addr.)
= 28 (byte addr.)

---

note

— array bound evaluation

C — static — eg) A[const-expr.];

C++ — static eval.
dynamic eval. — new char[size];

Java — dynamic eval. — int[] A = new int[10];

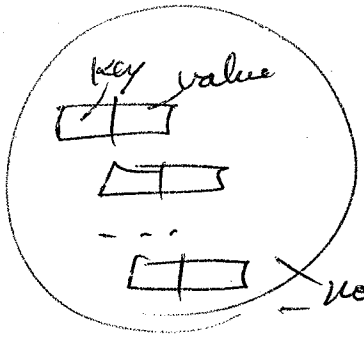
PASCAL — static eval. — A[min .. max]: ...
(min, max: constant)

ALGOL60 — evaluation upon procedure entry

eg) var A: array[(if (C < ∅) then 2 else 1) .. 20] of ---

§6.6 Associative arrays

- Perl — hash
- Python — dictionary
- C++ — unordered_map
- ..



— not ordered.
— search is done with hashing O(1)

Perl example

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

hash var (%...)

```
$salaries{"Perry"} = 58850;
```

— assignment, or new add.

Scalar var ($...) — key — value — value — element

```
delete $salaries{"Gary"};
```

— remove an element.

Scalar var

```
@salaries = ();
```

— entire hash is emptied

— Search for a key:

```
if (exists $salaries{"Shelly"}) ----
```

→ returns T/F.

(*) — Assign C++ unordered_map program -- or python dictionary.

— reading assignment — selected book (11th)

- §6.7 — Record types
- §6.8 — Tuple types
- §6.9 — List types
- §6.10 — Union types

Park

Python — dictionary

```
##################################
### To run, $>python p1.py p1-data
##################################

#!/usr/bin/env python
import os, sys, string

name_list = []  #empty list
mydictionary1 = {} #empty dictionary for <name, salary>
mydictionary2 = {} #empty dictionary for <salary, [name_list]>

for i in open(sys.argv[1]): #read input file
    (col1, col2) = i.strip().split()

##build dictionary1
    if mydictionary1.has_key(col1):
        mydictionary1[col1] = mydictionary1[col1] + int(col2)
    else:
        mydictionary1[col1] = int(col2)

##display dictionary1
keys = mydictionary1.keys()     #a list of keys
values = mydictionary1.values() #a list of values
indx = 0
for i in mydictionary1:
    print keys[indx]+'\t'+str(values[indx])
    indx = indx + 1


#####
print "----- round 2 ----------------"

for i in open(sys.argv[1]): #read input file
    (col1, col2) = i.strip().split()

##build dictionary2
    if mydictionary2.has_key(col2):
        mydictionary2[col2].append(col1)
    else:
        name_list.append(col1)
        mydictionary2[col2] = name_list
        name_list = [] ##flush name_list

##display dictionary2
keys2 = mydictionary2.keys()     #a list of keys
values2 = mydictionary2.values() #a list of values
indx = 0
for i in mydictionary2:
    for j in values2[indx]:
        print j + ","
    print str(keys2[indx])
    print "------------"
    indx = indx + 1
```

(Sample)

```
--- input data file ----
Name1   1000
Name2   2000
Name3   3000
Name4   1000
Name5   1500
Name6   2000
Name7   3500
Name8   3000
Name9   4000
Name10  2000
--------------------------

--- output -----
Name10  2000
Name6   2000
Name7   3500
Name4   1000
Name5   1500
Name2   2000
Name3   3000
Name1   1000
Name8   3000
Name9   4000
----- round 2 ------------
Name7,
3500
------------
Name5,
1500
------------
Name9,
4000
------------
Name2,
Name6,
Name10,
2000
------------
Name3,
Name8,
3000
------------
Name1,
Name4,
1000
------------
```

— structure allocation — similar to arrays

(C/C++ struct → can be allocated in stack or heap
(Java class → heap.
  object

ex) C like

```
struct point
{ int x;
  int y;
  char z;
};

point * p = new point();
```

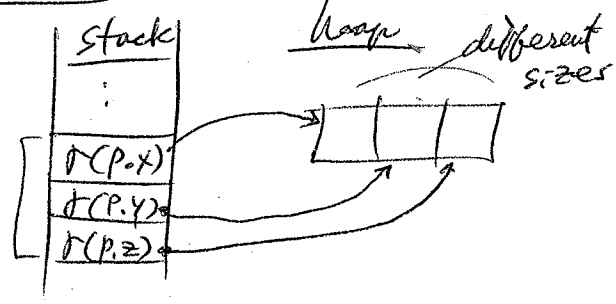
optional

```
p->x = 1;
p->y = p->x + 1;
```

or (*p).x

(*p).y (*p).x

allocation



— heap memory management

ex) C/C++

new / delete — explicit (instructions) (C/C++, Ada, ---)

heap alloc heap dealloc.

(finds the smallest addressed contiguous block of locations in the heap that are unused.

(Languages that require the program to manage pointers to allocate dynamic heap data structures also expect the program to restore heap blocks after done.

ex) C/C++

```
int *p;
p = new int;
delete p;
```

dynamic var

```
int *p;
p = new int [50];
delete [] p;
```

dynamic array

ex) Java — implicit (not by prog.) (Java, LISP, ---)

(# pointers
(objects are in heap

(automatic garbage collection by system.

— pointer and reference types — (Sebesta — §6.11)

pointer (type) — has a fixed size in memory regardless of the type point to.

— indirect access to elements of a known type.

efficiency — only pointer moves instead of large data structure moves.

dynamic data structure

```
int *p;
p = new int[5];
```

— operations on pointers

dereferencing — C/C++: *p — indirect access

→) C/C++

```
int *p, x;
p = &x;        —— p is a pointer to x
*p = *p+1;  ⇒ x=x+1
```

dereferencing op

→)

```
int *p, *z, x;
x=5;
p=&x;
*p = *p+1;
z=p;
printf(---, x);   — 6
printf(---, *p);  — 6
printf(---, *z);  — 6
```

— arrays and pointers in C

```
int a[n+1];
a[0] = x;  i=n;
while (a[i] != x)
    --i;
return i;
```

— array name $a \equiv \&a[0]$

⇒ using pointer:

```
int a[n+1]; int *p;
a[0]=x;  p=a+n;
while (*p != x)
    --p;
return (p-a);
```

$p = \&a[n] = \&a[0]+n$, $p-1 = \&a[n-1]$

if $p \to a[i]$, then $(p-1) \to a[i-1]$

index — if $p \to a[i]$, then $p=a+i$. ∴ $i = p-a$.

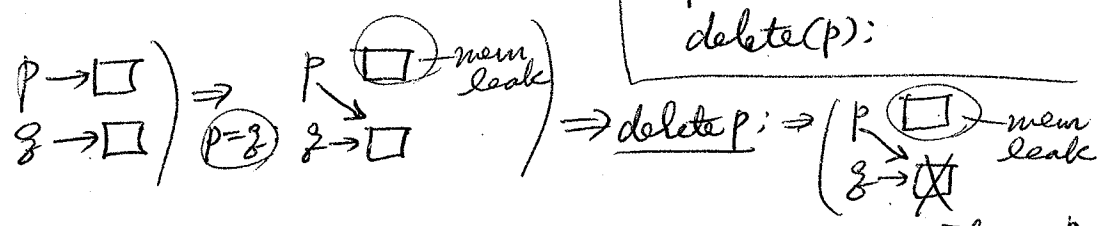
— pointer types

— Dangling pointer / memory leaks

ex) C/C++ delete p; — leaves p dangling
pointer

```
struct node
{ int value;
  node* next;
};
node* p, *q;
p=new node;
q=new node;
p=q;
delete p;
```

```
~~Java~~
class node
{ int value;
  node next;
};
node p, q; — objects
p=new node();
q=new node();
p=q;
delete(p);
```



dangling pointers: p, q

— memory deallocation
heap

- by explicit mstr. — ex). delete p; — C/C++, Ada, ...
- implicit (garbage collection) — LISP, Java,
  - → 2 approaches
    - Lazy approach: wait until all memory runs out, then collect garbage cells (mem. leaks)
    - Eager approach: use reference counter +, − when ref-counter = ϕ, return node to free list.

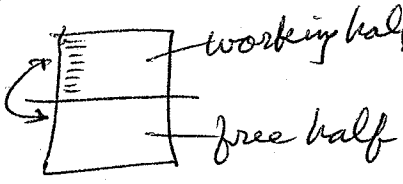
↓

garbage collection

Lazy approaches

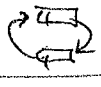
- Adv: saves time of frequent pointer operations
- dis: when garbage collection, all other works halt.

1. 2phase approach (mark/sweep)
   - mark all reachable nodes startup from all pointers (in stack).
   - Search entire memory (heap) for unmarked cells; and return them to the head of free list (LIFO) head.
2. Copy collector



- when working half is full, only reachable nodes are copied into free half and change the roles.

Eager approach

- as soon as having a memory leak, return it to the head of free list.
- 4) reference counting method.
  - adv: dynamic activation when ref-count = ∅. system doesn't halt.
  - dis: cannot detect isolated circular chain; 
    - reference counter consumes memory;
    - frequent pointer (reference) operations.

---

ref

Java

- system.gc(): programmer can invoke garbage collector.
- and, whenever system is idle, garbage collector is activated automatically.

— mark/sweep method (lazy approach)

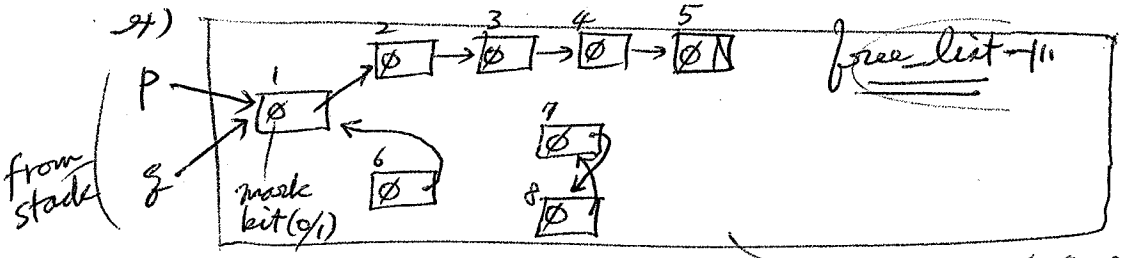
ex) q = new node(); ⟹

i.e.
```
class node
{ ≡
}
node q = new node();
```

if (free_list == null)
  activate mark/sweep garbage collector;
q = free_list; // get the head node from the free list.
free_list = free_list.next; update free list

1. mark phase:
   from any pointer (AR of func stack), mark all reachable nodes in the heap.
2. sweep phase:
   starting from the lowest mem. address,
   scan all memory (heap);
   if a node is unmarked, return it to the head of free list.

ex)



Layout in mem.

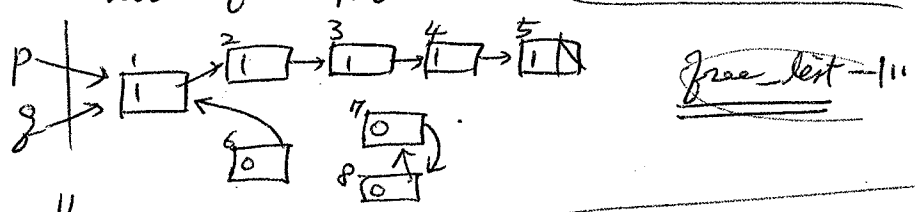
	1	2	3	4	5	6	7	8
key	x	x	x	x	x	x	x	x
next	2	3	4	5	-1	1	8	7

garbages

p(L1) head index — 1
q(L2) head index — 1

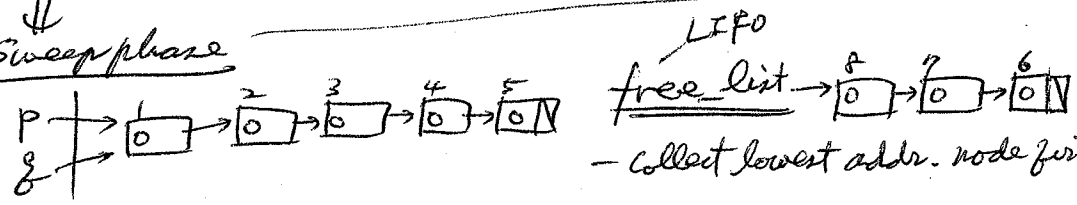
⇓

mark phase
(mark all reachable nodes from p, q.



⇓

Sweep phase

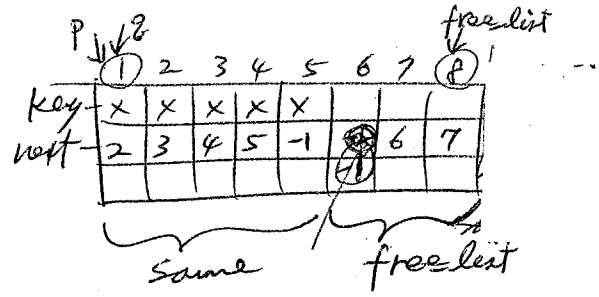


— collect lowest addr. node first

↓

↓

final configuration (mem_layout) :



p (L1) head indx — 1
q (L2) head indx — 1
free_list head indx — 8

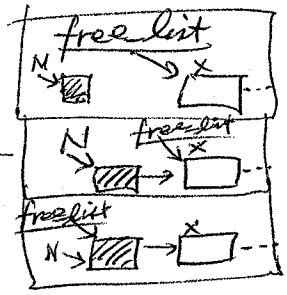
— implementation (mark/sweep)

```
— mark(R)        reference (pointer) or (mem-indx)
{ if (R.MB == ϕ)        mark bit
      R.MB = 1 ;
  if (R.next != null)
      mark(R.next);
}
```

— mark all reachable nodes from pointer R. by calling mark func recursively.

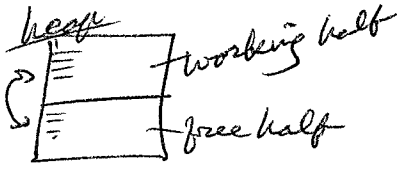
```
— sweep()
{ i = h;        heap starting addr.
  while (i ≤ n)        mem (heap) bound
  { if (i.MB == ϕ) // if unreachable
        free(i); // return (attach) it to the head of free_list.
    else
        i.MB = ϕ; // clear marked bit
    i = i+1 ;
  }
}
```

```
— free(N)        reference (addr.)
{ N.next = free_list;
  free_list = N;
}
```

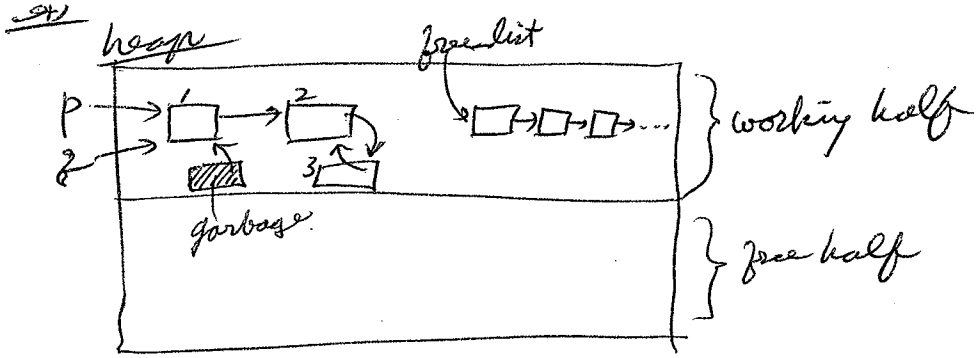


//

— Copy collector method (lazy approach)

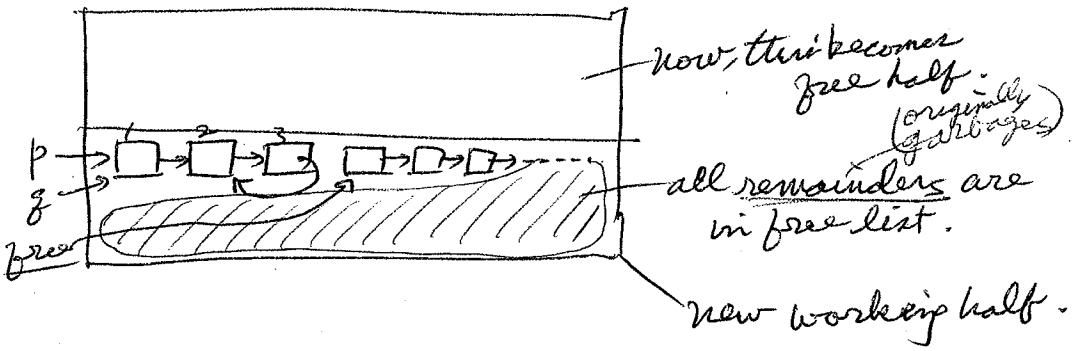


— when working half is full,
only reachable nodes (from any pointers)
are copied to free half, and change roles



assume, working half is full now. (i.e., free list → 11.)
empty

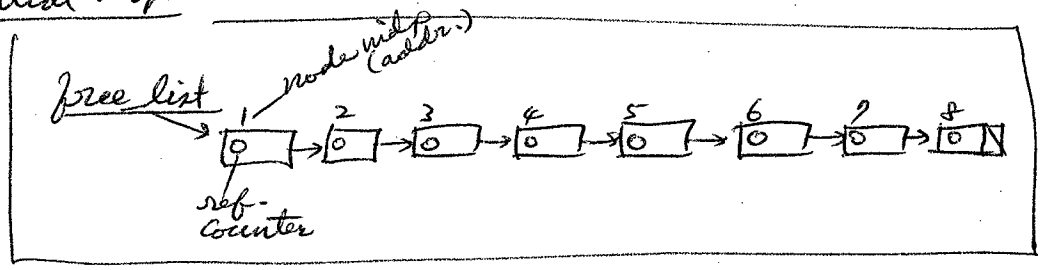
⇓



— Reference Counting method (Eager approach)

reference counter is updated when new/delete or pointer assign.

94) initial heap



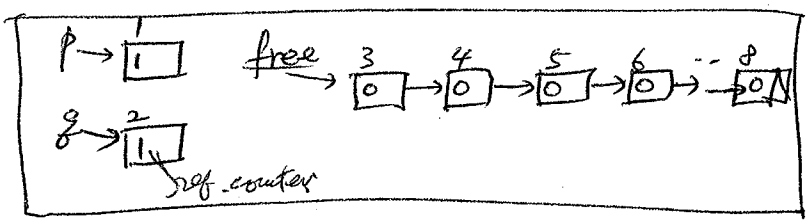
— free list — LIFO (stack like)

↓

↓

```
p = new node();
q = new node();
```

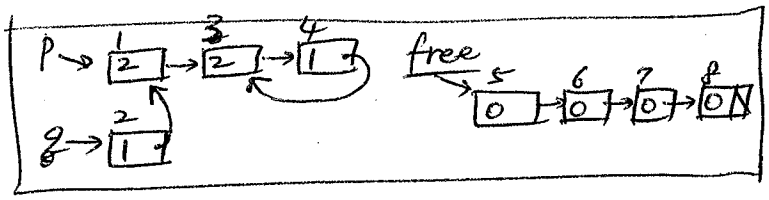
⇒



↓

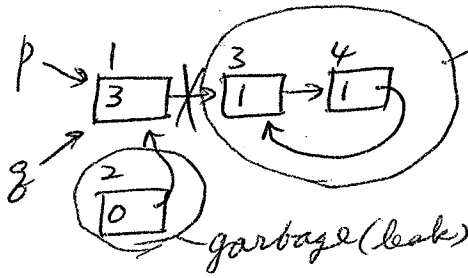
```
p->next = new node();
p->next->next = new node();
p->next->next->next = p->next;
q->next = p;
```

⇒



↓

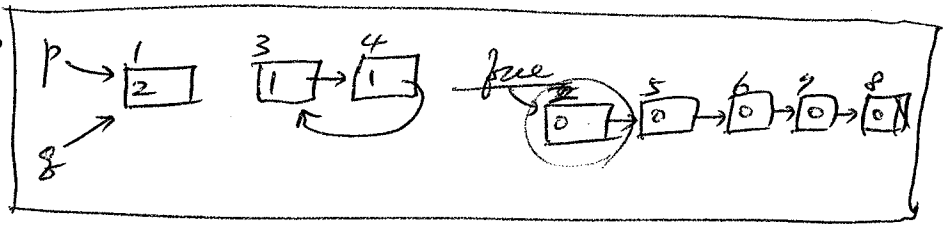
```
p->next = null;
q = p;
```



isolated circular chain

problem:
cannot return these to the free list.
∵ reference counter ≠ ∅.

⇒



adv: dynamic activation when ref_count = ∅;
System doesn't halt during garbage collection.

dis: cannot detect isolated circular chain.
reference counter consumes memory
frequent pointer operations