Data Types, Memory

Data Types

- Values held in machine locations
- Integers, reals, characters, Booleans are built into languages as primitive types
  - Machine location directly contains the value
  - Efficiently implemented, likely understood by the instruction set
- Others built on top of them : structured types
  - Laid out in sequence of locations in the machine
  - Arrays, records, pointers.
  - Hopefully can be treated as **first class citizens**
    - A first class citizen can be passed as a parameter, returned from a subroutine, or assigned into a variable.

Data Types

- What are types good for?
  - implicit context
  - checking - make sure that certain meaningless operations do not occur
    - type checking cannot prevent all meaningless operations
    - It catches enough of them to be useful
- Polymorphism results when the compiler finds that it doesn't need to know certain things

Data Types

- STRONG TYPING has become a popular buzz-word
  - like *structured programming*
  - informally, it means that the language prevents you from applying an operation to data on which it is not appropriate
- STATIC TYPING means that the compiler can do all the checking at compile time

Type Systems

- Examples
  - Common Lisp is strongly typed, but not statically typed
  - Ada is statically typed
  - Pascal is almost statically typed
  - Java is strongly typed, with a non-trivial mix of things that can be checked statically and things that have to be checked dynamically

Type Systems

- Common terms:
  - discrete types – countable
    - integer
    - boolean
    - char
    - enumeration
    - subrange
  - Scalar types - one-dimensional
    - discrete
    - real

Type Systems

- Composite or structured types:
  - records (unions)
  - arrays
    - strings
  - sets
  - pointers
  - lists
  - files

Variant Records and Unions

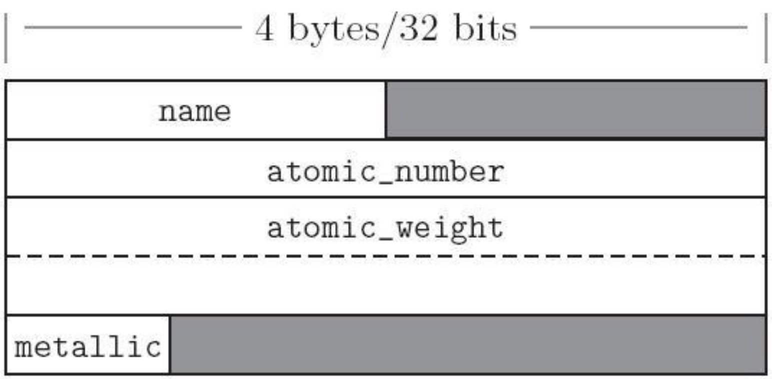
- Back when memory was scarce…
  - Variant records allowed two or more different fields to share the same block of memory
  - Called Variant in Pascal, Union in C

```
union myUnion {
        int i;        // 32 bits of storage
        float f;      // Same 32 bits of storage
};

Union myUnion u;

u.i  accesses storage as Integer
u.f accesses storage as float
```

How might we do something in Java that allows accesses to a value that might be of different types?

Records (Structures)

- Memory layout and its impact (structures)




```
struct element {
  char name[2];
  int atomic_number;
  double atomic_weight;
  bool metallic;
}
```

Figure 7.1: **Likely layout in memory for objects of type** `element` **on a 32-bit machine.** Alignment restrictions lead to the shaded "holes."

Type Systems

- ORTHOGONALITY is a useful goal in the design of a language, particularly its type system
  - A collection of features is orthogonal if there are no restrictions on the ways in which the features can be combined (analogy to vectors)
- For example
  - Pascal is more orthogonal than Fortran, (because it allows arrays of anything, for instance), but it does not permit variant records as arbitrary fields of other records (for instance)
- Orthogonality is nice primarily because it makes a language easy to understand, easy to use, and easy to reason about

Type Checking

- A TYPE SYSTEM has rules for
  - type equivalence (when are the types of two values the same?)
  - type compatibility (when can a value of type A be used in a context that expects type B?)
  - type inference (what is the type of an expression, given the types of the operands?)
- Type compatibility / type equivalence
  - Compatibility is the more useful concept, because it tells you what you can DO
  - The terms are often (incorrectly, but we do it too) used interchangeably.

Type Equivalence

- Sometimes we need to know when two types are equivalent, but this can be trickier than it sounds

```
struct complex {
     float re, im;
};
struct polar {
     float x, y;
};
struct {
     float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[10];
// which are equivalent types?
```

Type Checking

- Two major approaches: structural equivalence and name equivalence
  - Name Equivalence
    - Two types are the same if they have the same name
  - Structural Equivalence
    - Two types are the same if they have the same structure
    - Structural equivalence depends on simple comparison of type descriptions substitute out all names
      - expand all the way to built-in types
  - Name equivalence is more fashionable these days

Type Checking

- Coercion
  - When an expression of one type is used in a context where a different type is expected, one normally gets a type error
  - But what about

```
var a : integer; b, c : real;
    ...
c := a + b;
```

Type Checking

- Coercion
  - Many languages allow things like this, and COERCE an expression to be of the proper type
  - Coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
  - Fortran has lots of coercion, all based on operand type

Type Checking

- C has lots of coercion, too, but with simpler rules:
  - all `float`s in expressions become `double`s
  - short `int` and `char` become `int` in expressions
  - if necessary, precision is removed when assigning into LHS
- In effect, coercion rules are a relaxation of type checking
  - Recent thought is that this is probably a bad idea
  - Languages such as Modula-2 and Ada do not permit coercions
  - C++, however, goes hog-wild with them
  - They're one of the hardest parts of the language to understand

Functions as Types

- Some languages allow functions to behave as “first class citizens”
  - Function can be treated like a data type or variable
  - Can pass a function as an argument
- Pascal example:
  - function newton(a, b: real; function f: real): real;
  - Know that f returns a real value, but the arguments to f are unspecified.

Java Example

```
public interface RootSolvable {
	double valueAt(double x);
}

public class MySolver implements RootSolvable
{
	double valueAt(double x)
	{
		…
	}
}

public double Newton(double a, double b, RootSolvable f)
{
		…
		val = f.valueAt(x);
		…
}
```

```
mysolver = new MySolver();
z = Newton(a,b,mysolver);
```

Not a true first-class citizen since a function can't be constructed and returned by another function

Arrays

- A sequence of elements of the **same** type stored consecutively in memory
- Element can be accesses quickly [O(1)]
- Accessed via indexing
  - A[i] : i→ index
- Index is often an integer but does not have to be
  - Must be efficiently computed
  - Here we are not including “associative” arrays that are really more like hash tables
- When is array bound computed?
- When is the space for the array allocated?
- Where is the space for the array allocated?
  - Java: from the Heap

Array Initialization

- Should the values in an array be pre-initialized?
  - Java initializes all values to 0 or null
  - C/C++ do no initialization, array contains whatever values happen to be sitting in memory
- Issue of efficiency

Arrays in Pascal

- May have any range of indices

  array [21-30] of real

- May have non integer indexes

```
array [(Mon,Tue, Wed,Thu,Fri)] of integer;
array [char] of token;
type token = (plus, minus, times, divide,number,
  lparen, rparen, semi);
```

- These non-integer values really map to integer values internally for efficiency purposes
  - E.g. Mon=0,Tue=1, Wed=2, etc.

Arrays

- Should array *type* include bounds?
- Pascal did and it causes some problems
  - typeof(A[10]) ≠ typeof(A[100])
- Function arguments with arrays are problematic
  - Sort function with an array size of size 10 can't take array of size 9
  - Instead must pass array bounds as parameters

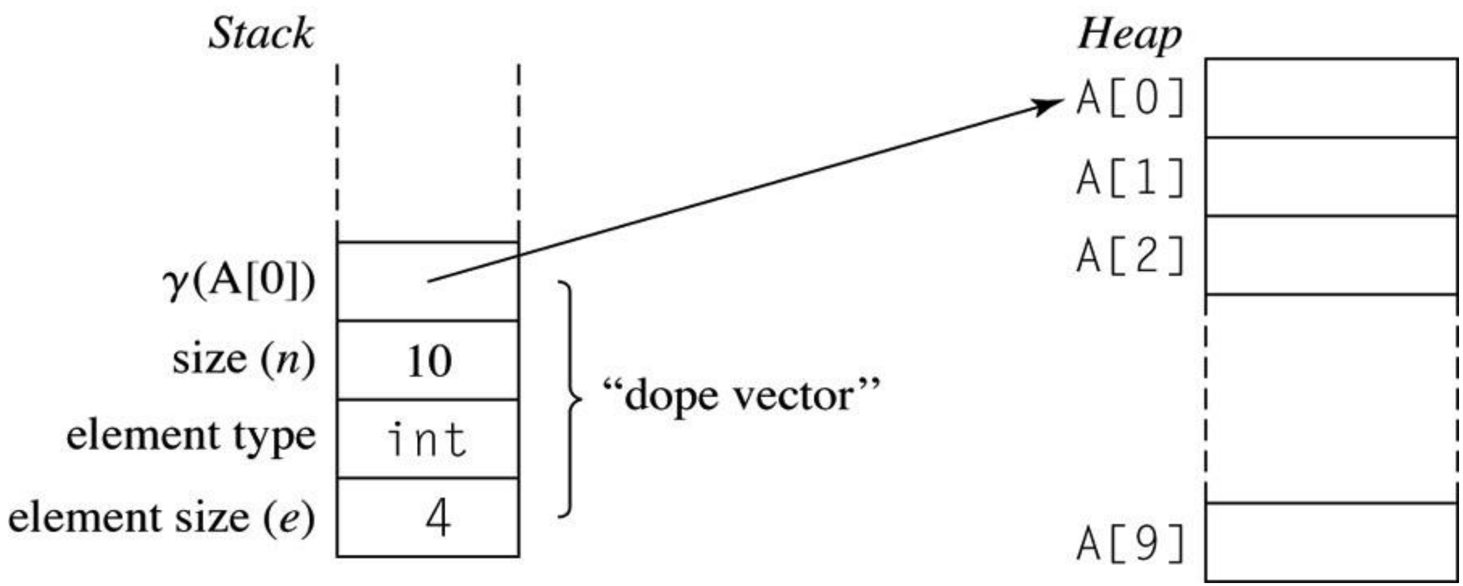
Arrays

Layout

- Determines the machine address of the i'th element relative to the address of the first element
- Different from allocation
  - Reserve actual machine memory for the array
- The elements of the array appear in consecutive locations

Arrays

Layout(C/Java-Like Language)



int[] A = new int[10];
γ(A[i]) = γ(A[0])+e*i
0 <= i < n

e=element size, i=index

Strongly typed language requires checking type in dope vector

Arrays

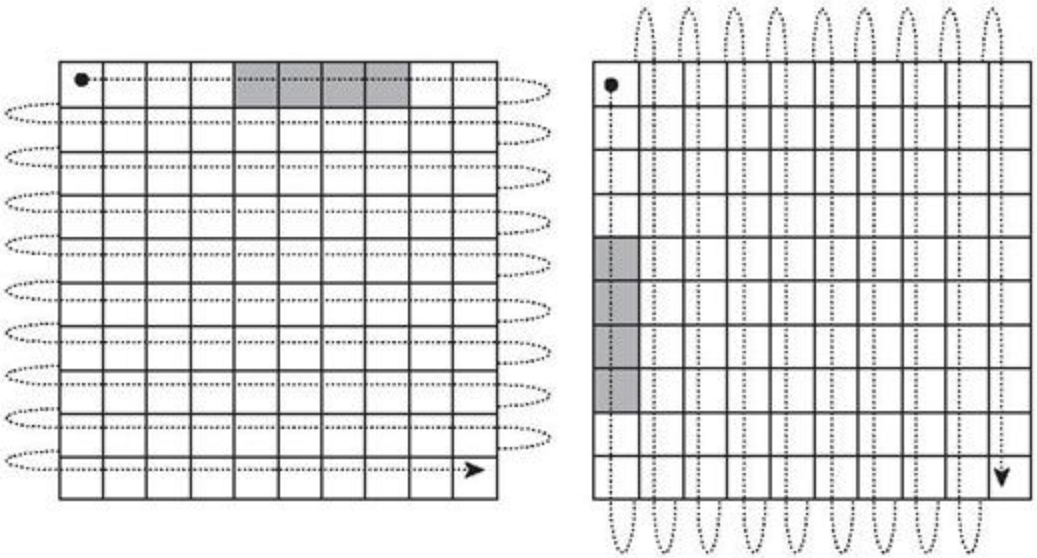
```
var  A : array [low .. high] of T
```

- base
  - Starting address of the first element A[low]
- width
  - size of an element of type T
- The elements are stored at
  - base, base+width, base + 2*width ....
- Address of A[i] computed in 2 parts
  - Compile time : offset from base
  - Run time : location of base

Arrays

- Address of A[i]
  = base +(i-low)*width
  = i*width + (base-low*width)
- (base-low*width) may be precomputed and stored
  - This is generally the value associated with an array variable
- i*width : must be computed at runtime
- If low = 0
  - Address of A[i] = i*width + base
- Time to compute the address is independent of i
  - So we get O(1) or constant access time

Arrays



Row-major order Column-major order

Row- and column-major memory layout for two-dimensional arrays. In row-major order, the elements of a row are contiguous in memory; in column-major order, the elements of a column are contiguous. The second cache line of each array is shaded, on the assumption that each element is an eight-byte floating-point number, that cache lines are 32 bytes long (a common size), and that the array begins at a cache line boundary. If the array is indexed from A[0,0] to A[9,0], then in the row-major case elements A[0,4] through A[0,7] share a cache line; in the column-major case elements A[4,0] through A[7,0] share a cache line.

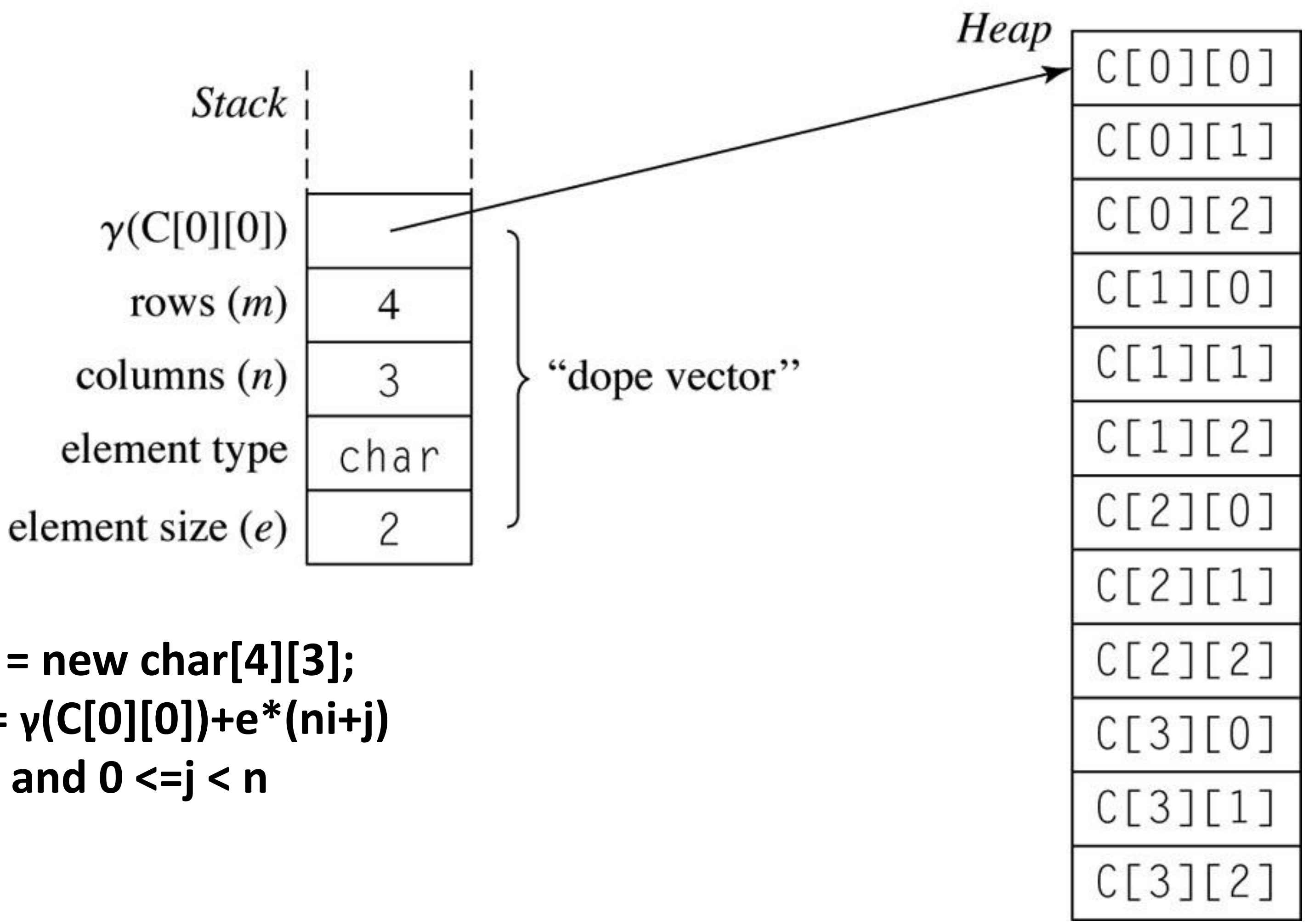
Multidimensional Arrays

- Common in all languages
  - C : A[200][200]
- Allocated in linear fashion
- Row major
  - Store by rows: row 1, row 2, row 3, ….
- Column major
  - Store by columns

Multidimensional Arrays

Layout(C/Java-Like)

Row major order



char[][] C = new char[4][3];
γ(C[i][j]) = γ(C[0][0])+e*(ni+j)
0 <= i < m and 0 <=j < n

Multidimensional Arrays

(j-low2)

b	b+1	b+2
b+3	b+4	b+5
b+6	b+7	b+8

(i-low1)

- Address of M[i][j]

  $$base + (i\text{-}low_1)*w_1 + (j\text{-}low_2)*w_2$$

  – w_1 : width of a row $= w_2 * n_2$

  – w_2 : width of an element

  – n_1 : number of elements in a column

  – n_2 : number of elements in a row $= high_2\text{-}low_2 + 1$

- Fixed part : $base - low_1 * w_1 - low_2 * w_2$
- Variable part : $i * w_1 + j * w_2$

Multi-D Arrays(Java)

- Java actually stores only 1D arrays; multi-dimensional arrays are references to other arrays

```
int[][] nums = new int[4][3];
```

Strings

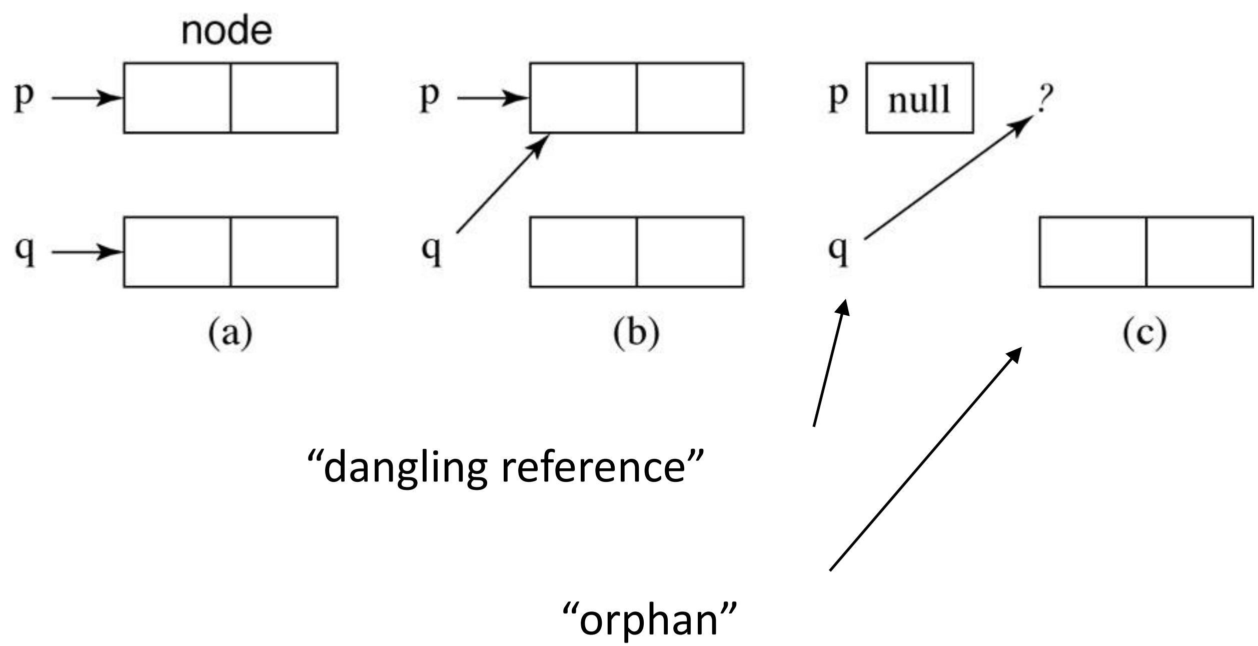
- Strings are typically just arrays of characters
- They are often special-cased, to give them flexibility (like polymorphism or dynamic sizing) that is not available for arrays in general
  - It's easier to provide these things for strings than for arrays in general because strings are one-dimensional and (more important) non-circular

Dangling Pointers

- Structures or Classes are often used as nodes within dynamic data structures, such as linked lists
- Raises the possibility of the **dangling pointer**
  - A pointer to storage used for another purpose and the storage is subsequently deallocated
- Garbage
  - Allocated but inaccessible memory locations
- Programs that create garbage are said to have *memory leaks*

Dangling Pointer Example

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



Memory Leak Terms

- Dangling reference/Widow
  - A pointer to storage used for another purpose and the storage is subsequently deallocated
- Garbage/Orphan
  - Allocated but inaccessible memory locations
- Programs that create garbage are said to have *memory leaks*

Avoiding Garbage

- Many languages ask the programmer to explicitly manage the heap, where memory is allocated
  - C, C++,…
  - User must make sure to destroy **everything** that is allocated
  - Memory management is generally not central to the problem the programmer is trying to solve
  - What if something is missed? Easy to do…

```
void foo()
{
  p = new node();
  if (b) return;
  delete(p);
}
```

- Interpreted and functional languages generally do automatic garbage collection
  - Java, C#, Lisp,…

Garbage Collection

- Motivation from functional programming
- Increased importance due to OOP

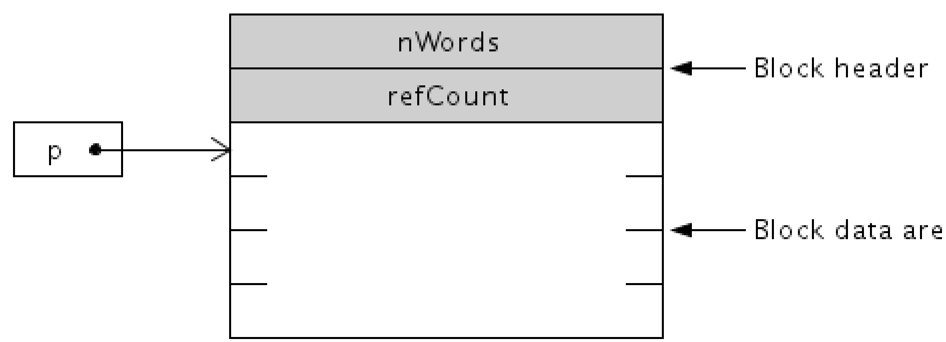
How do we reduce/eliminate the burden of memory management from the programmer?

Garbage Collection Algorithms

- Reference counting
- Mark-Sweep
- Copy collection
- In Java
  - The garbage collector runs as a low-priority thread. It is automatic but it can be explicitly called by: System.gc() (regardless of the state of the heap at the time of the call).

Garbage Collection
Reference Counting

- Free List
  - Heap is a continuous chain of nodes called the free list
    - Implemented various ways, we'll skip implementation
  - Each node has an extra field to keep a count as well as a field to keep track of the node size
- Reference Count
  - Number of pointers referencing that node
  - Initially set to 0



Garbage Collection
Reference Counting

- Node creation via new()
  - Get nodes from the free list
  - Set reference count to 1
- Pointer Assignment
  - e.g. p=q;
  - Increment the reference count of q by 1
  - Decrement the reference count of p by 1
    - If zero, nothing references p so it is safe to delete
      - must also decrement reference count for any pointer in p’s data area by one. If one of these counts becomes zero, repeat for it’s descendants
      - Destroy p
  - Then perform the assignment

Garbage Collection
Reference Counting

- Pointer Deletion
  - e.g. delete p;
  - Decrement p's reference count
    If refcount == 0
        For every pointer q in p's data area
            delete q
        Put p on the free list

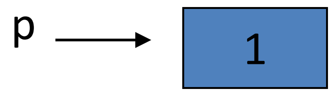
    Set p to null

Garbage Collection
Reference Counting

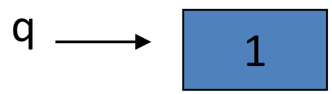
- The algorithm is activated dynamically on
  - new
  - Delete
  - assignment
- Advantages
  - Very simple, fast, non-compacting garbage collection
  - Heap maintenance spread throughout program execution (instead of suspending the program when the garbage collector runs)
  - Must not forget to adjust reference counts on any pointer assignment (including passing pointers as subroutine arguments), or disaster can happen

Reference Counting Example

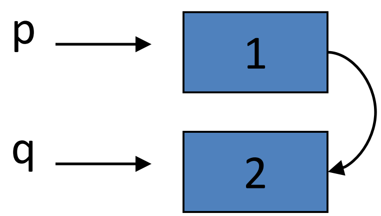
```
node p, q, t;
p = new node();
```




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

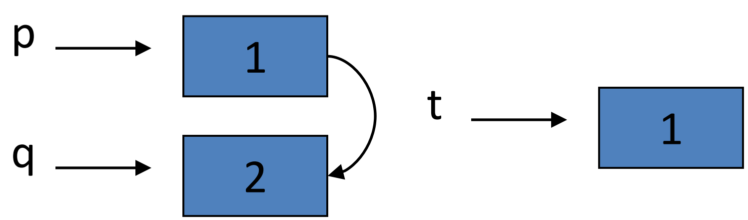



```
p.next = q;
```

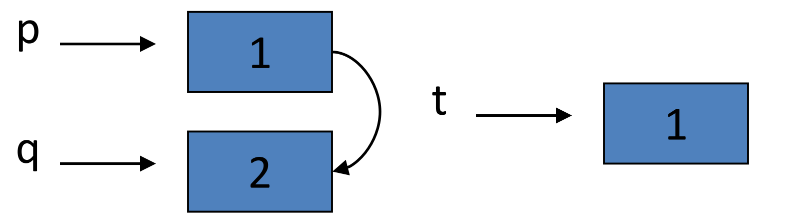


if p.next pointed to something, we'd decrement the ref

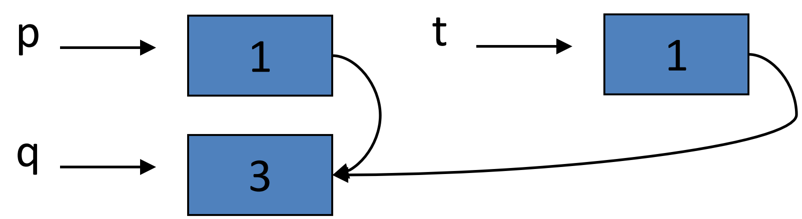
```
t = new node();
```



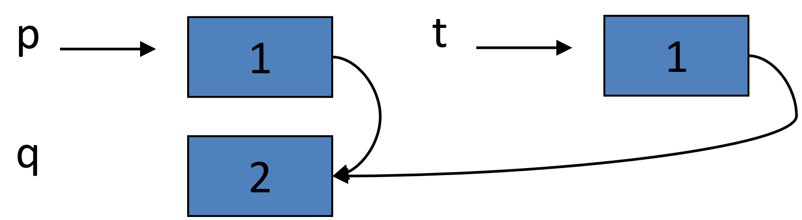
Reference Counting Example



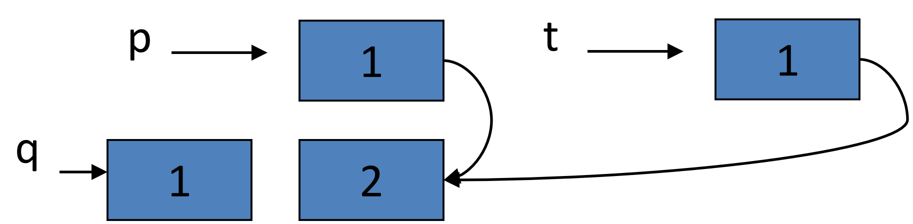

```
t.next = q;
```



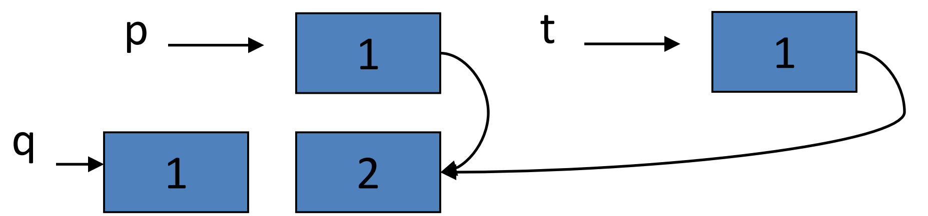

```
delete q;
```



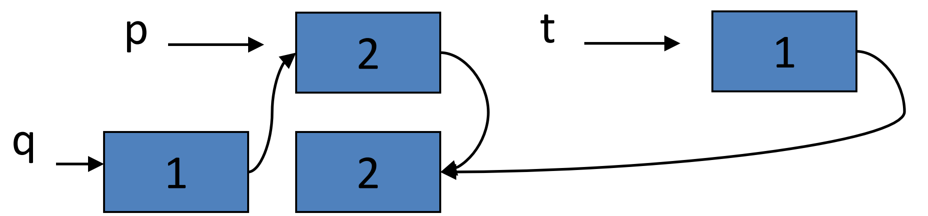

```
q = new node();
```



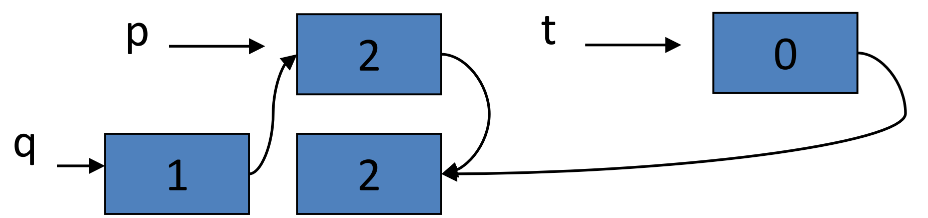
Reference Counting Example

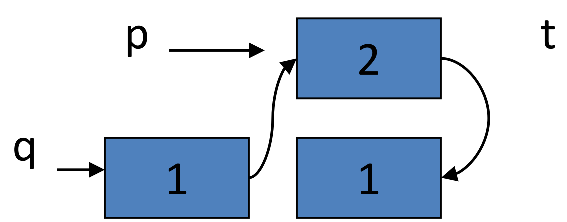



```
q.next = p;
```

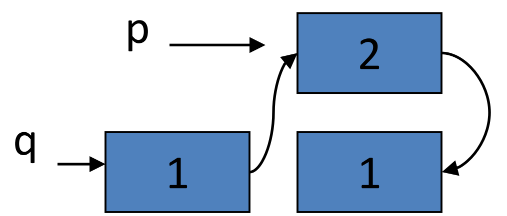



```
delete t;
```

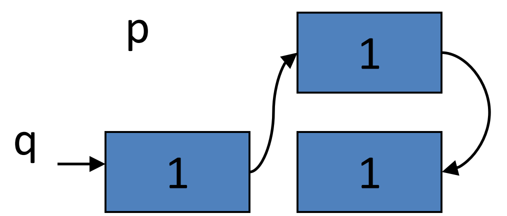




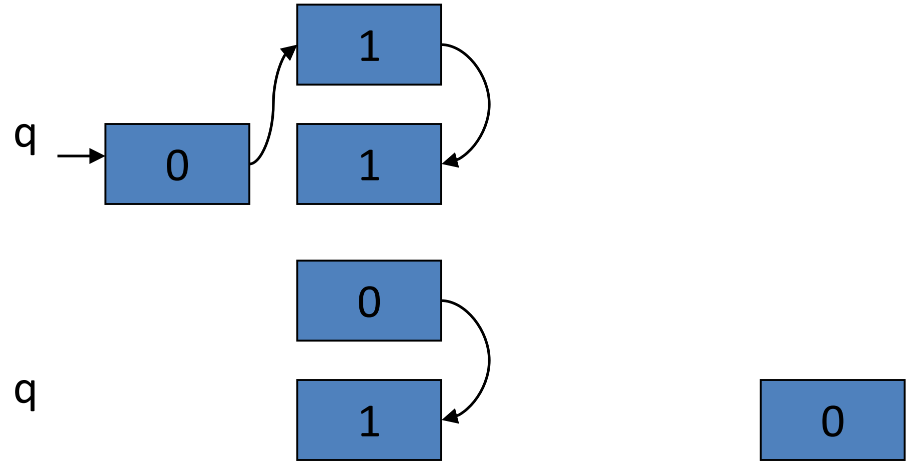
Reference Counting Example




```
delete p;
```

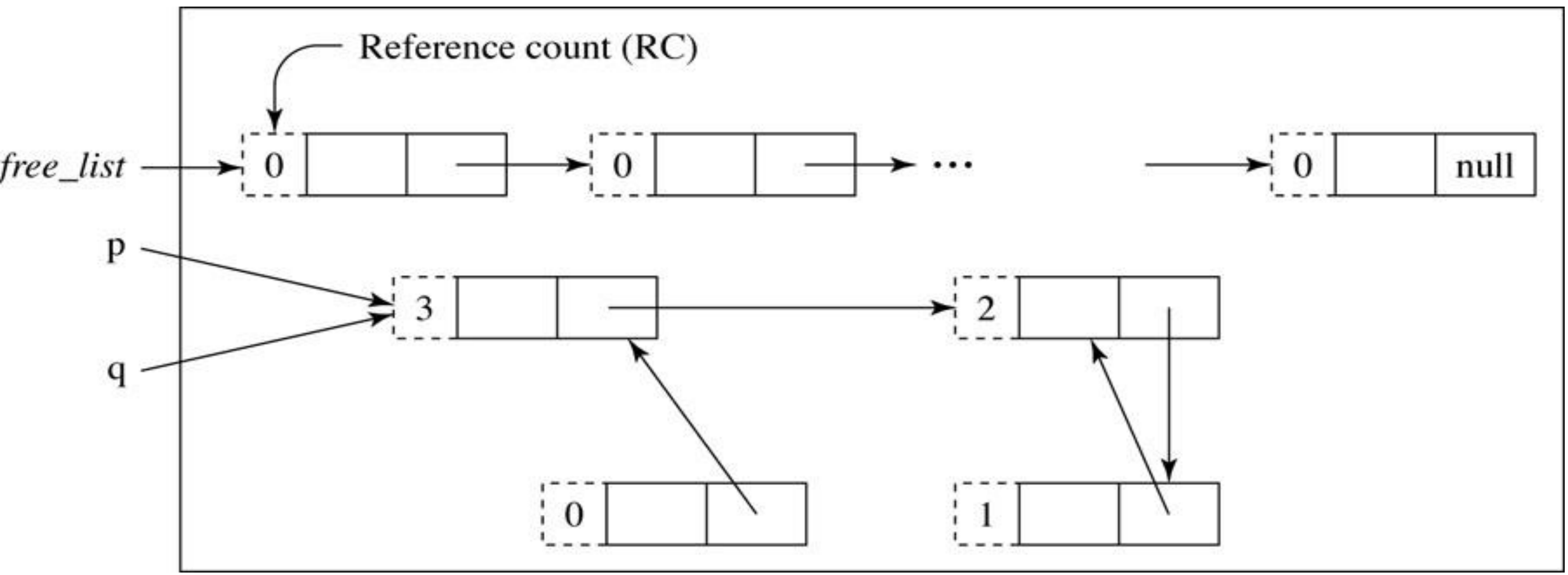



```
delete q;
```



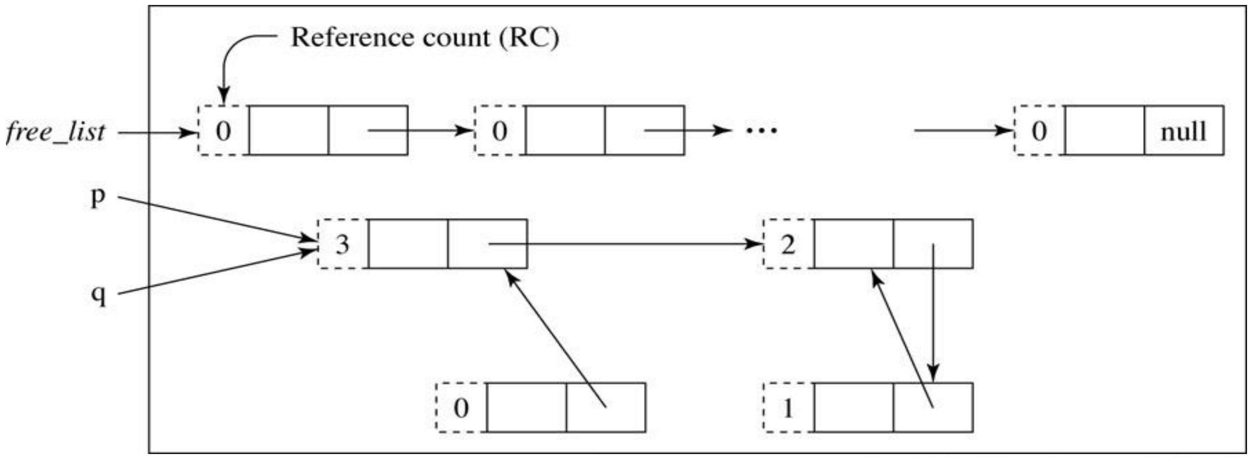
Garbage Collection

Reference Counting

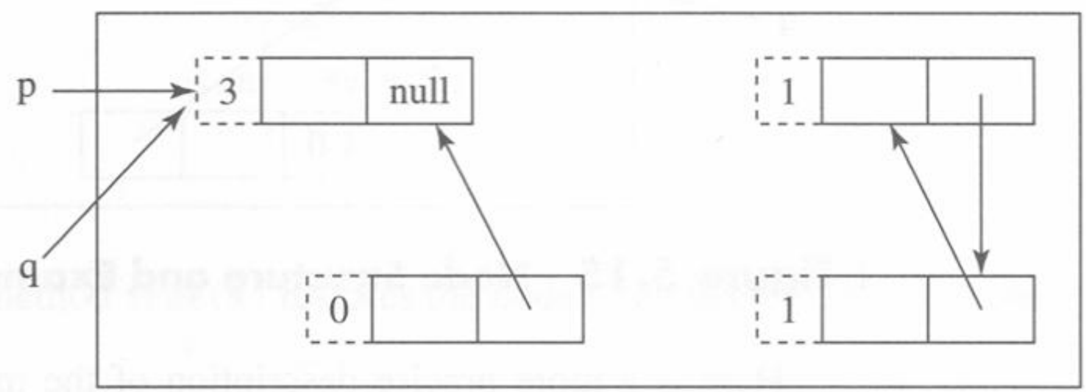


Reference Counting

- Minor Problem – Storage overhead for reference count
- Major Problem - Can’t handle circular chains of nodes



p.next=null;



Garbage Collection

Mark-Sweep

- Unlike reference counting, called when the heap becomes full
  - i.e. free list becomes empty
- Orphans are reassigned to the free list
  - Possibly large number of nodes
  - May be time consuming
  - Advantage over reference counting is it reclaims all garbage, even those in circular chains
- 2 Pass algorithm
  - 1st pass: Mark all the nodes if they are accessible
  - 2nd pass: Reassign the orphans

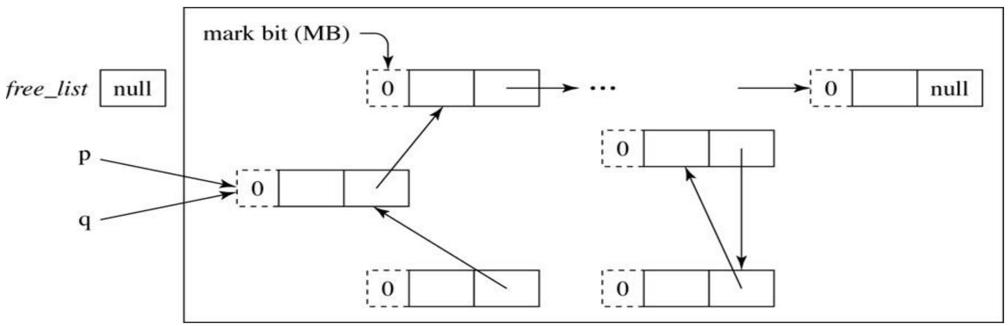
Garbage Collection

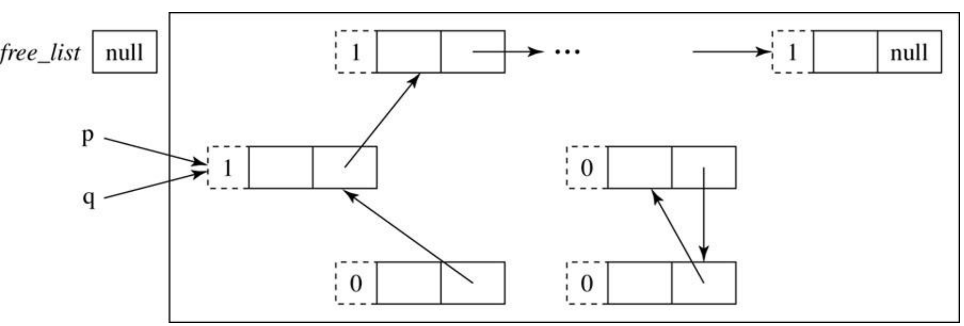
Mark-Sweep

- Mark Phase
  - Start with the active variables
  - Follow the links and “mark” the nodes that can be accessed
  - All unmarked nodes are orphans
- Sweep Phase
  - Follow all nodes in the heap
  - If the node is unmarked return to free list
  - Unmark all nodes that were not returned

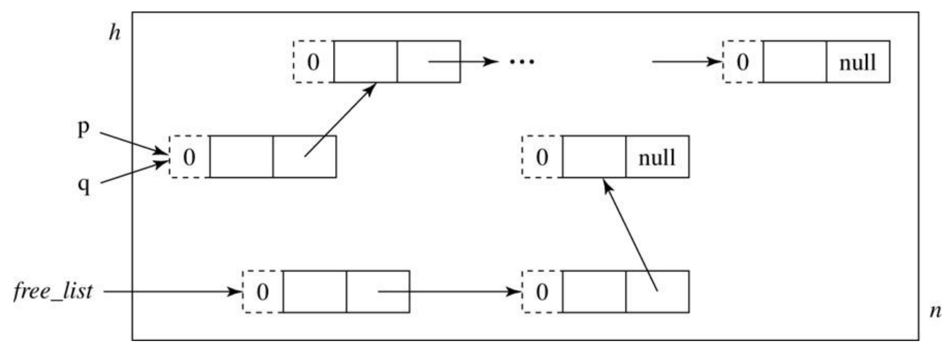
Garbage Collection

Mark-Sweep





After Mark Phase



After Sweep Phase

Online Demo

- Heap of Fish
- http://www.artima.com/insidejvm/applets/HeapOfFish.html

Garbage Collection

Mark-Sweep

- Advantages
  - Not invoked unless needed
    - Small programs don’t need it
    - Typically perform a large number of new/delete before this is needed
  - Reclaims all garbage
    - No problem with circular chains
  - Reduced memory overhead
    - Integer vs. a bit
- Disadvantages
  - Time consuming when used
    - 2 pass algorithm

Garbage Collection

Copy Collection aka Stop and Copy

- Time-space compromise compared to Mark-Sweep
- Also invoked only when heap becomes full
- Significantly faster than Mark-Sweep
  - Only 1 pass over the heap
  - But heap size is effectively reduced by half
    - i.e. copy collection uses a lot more memory, (but this is not as bad as it sounds if using virtual memory, can still have data in all available physical memory)

Garbage Collection

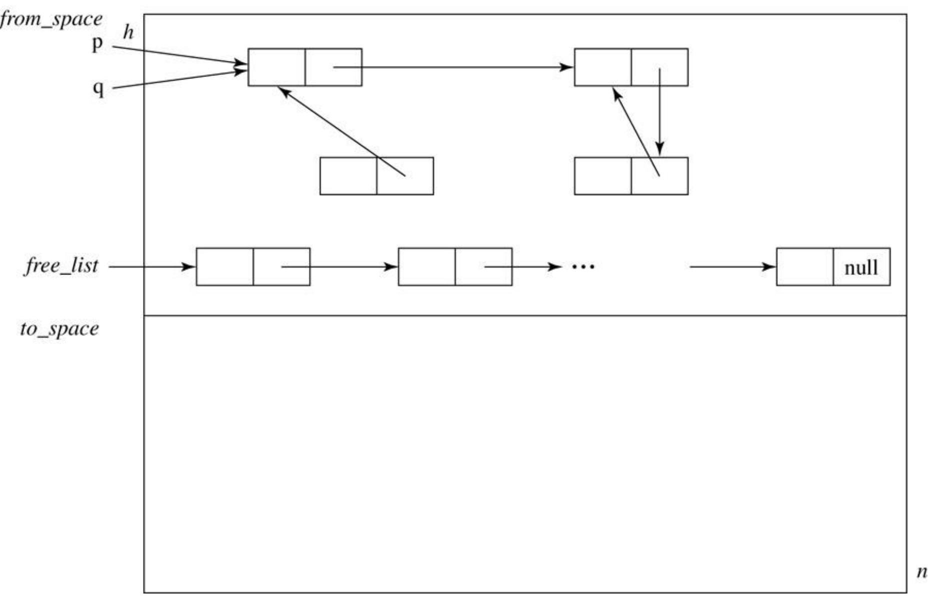
Copy Collection

- Divide the heap into two equal halves
  - *from_space*: All active nodes are kept here.
  - *to_space*: Used as a copy buffer

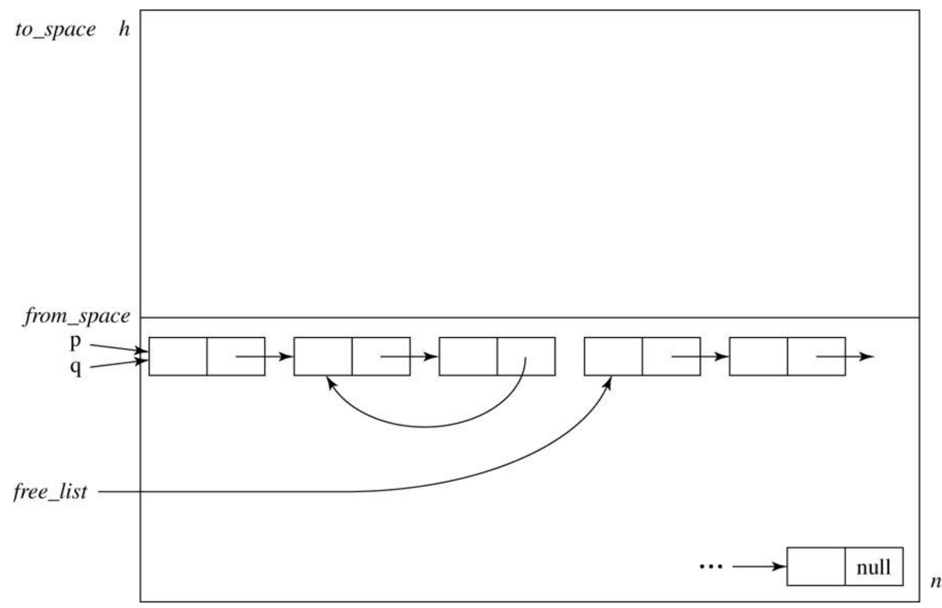
  When the *from_space* becomes full
  - All accessible nodes are copied into *to_space*
    - The descendents are copied as well
    - Copying to the to_space called *Forwarding*
    - Everything in the *from_space* is then added to the free list
  - Swap the roles of *from_space* and *to_space*
  - Eliminates the inaccessible nodes
  - Skipping some details here of allocating nodes from the free list of the to_space

Garbage Collection

Copy Collection



Initial Heap Organization



After Copy Collection Activation

Efficiency of Copy Collection vs. Mark Sweep

- M = heap size
- R = amount of live memory
- r = R/M is the residency
- m = amount of memory reclaimed
- t = time needed for reclaiming memory
- e =m/t is the efficiency of garbage collection (memory reclaimed per time)

Efficiency Continued

- Comparison:

$$t_{copy} = aR \qquad t_{MS} = bR + cM$$

$$m_{copy} = \frac{M}{2} - R \qquad m_{MS} = M - R$$

$$e_{copy} = \frac{M}{2aR} - \frac{1}{a} = \frac{1}{2ar} - \frac{1}{a}$$

$$e_{MS} = \frac{M - R}{bR + cM} = \frac{1 - r}{br + c}$$

Since r < 1, copy collection better for small r

As r increases, mark sweep becomes more efficient (as r approaches M/2)

Garbage Collection Today

- Many newer, complex algorithms proposed
- Active area of research
  - Incremental garbage collectors
  - Efficient garbage collectors (e.g., no recursion)
  - Generational garbage collectors
    - Separate objects that are in a young/old generation; older are more likely to survive, so might only scan younger generations, condemn older generations less frequently
- Hard to judge algorithm in isolation
  - Often must consider hardware considerations such as paging, virtual memory