GARBAGE COLLECTION METHODS

Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
e-mail: hjs@umiacs.umd.edu

GARBAGE COLLECTION

- Management of storage allocation from a heap (e.g., strings, lists, etc.)
- Allocation is usually explicit (e.g., CONS in LISP, NEW in PASCAL)
- Deallocation is explicit or implicit
  1. PL/1: explicit via FREE
  2. LISP: implicit via garbage collection
     - process of determining what part of storage is no longer being referenced and returning it to the available pool
- Ex:

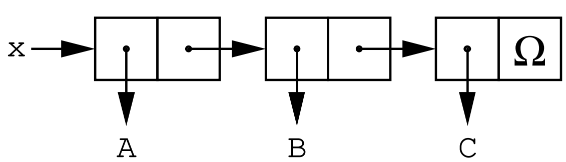
```
procedure reverse(x);
if empty(x) then x
else reverse2(x,Ω);

procedure reverse2(x,y)
if empty(x) then y
else reverse2(rest(x),add(first(x),y));
```

- `add(x,y)` grabs a 2-element cell from AVAIL and yields $\boxed{x \mid y}$

`reverse[(A B C)]` ⇒ `reverse2[(A B C),Ω]`

y←Ω



GARBAGE COLLECTION

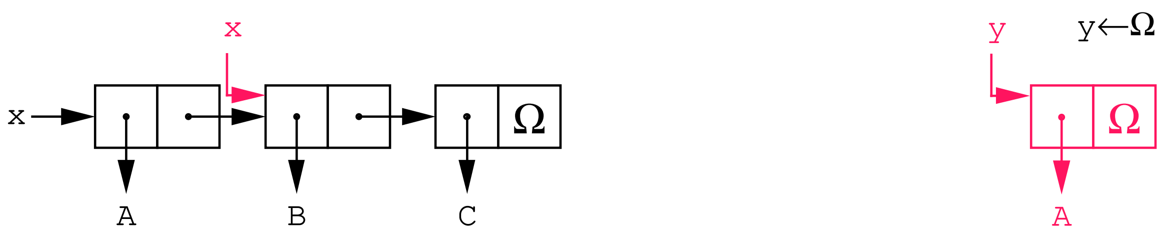
- Management of storage allocation from a heap (e.g., strings, lists, etc.)
- Allocation is usually explicit (e.g., CONS in LISP, NEW in PASCAL)
- Deallocation is explicit or implicit
  1. PL/1: explicit via FREE
  2. LISP: implicit via garbage collection
     - process of determining what part of storage is no longer being referenced and returning it to the available pool
- Ex:

```
procedure reverse(x);
if empty(x) then x
else reverse2(x,Ω);

procedure reverse2(x,y)
if empty(x) then y
else reverse2(rest(x),add(first(x),y));
```

- `add(x,y)` grabs a 2-element cell from AVAIL and yields | x | y |

`reverse[(A B C)]` $\Rightarrow$ `reverse2[(A B C),Ω]`



GARBAGE COLLECTION

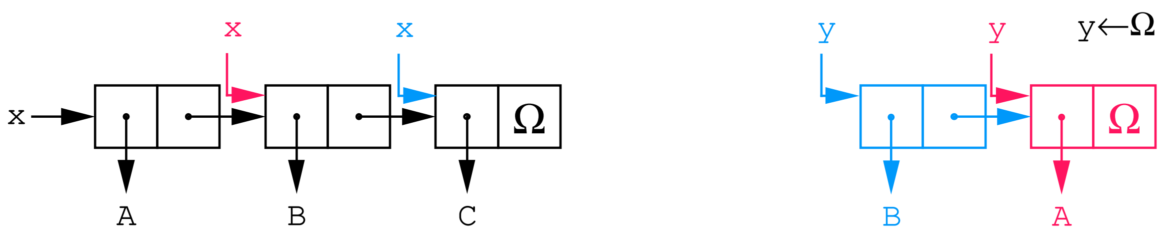
- Management of storage allocation from a heap (e.g., strings, lists, etc.)
- Allocation is usually explicit (e.g., CONS in LISP, NEW in PASCAL)
- Deallocation is explicit or implicit
  1. PL/1: explicit via FREE
  2. LISP: implicit via garbage collection
     - process of determining what part of storage is no longer being referenced and returning it to the available pool
- Ex:

```
procedure reverse(x);
if empty(x) then x
else reverse2(x,Ω);

procedure reverse2(x,y)
if empty(x) then y
else reverse2(rest(x),add(first(x),y));
```

- `add(x,y)` grabs a 2-element cell from AVAIL and yields $\boxed{x \mid y}$

`reverse[(A B C)]` ⇒ `reverse2[(A B C),Ω]`



GARBAGE COLLECTION

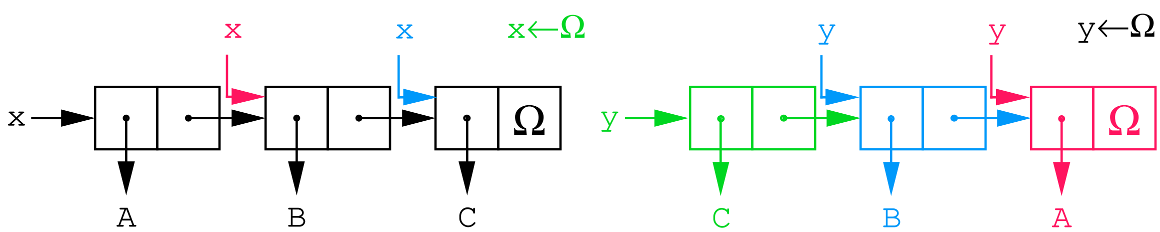
- Management of storage allocation from a heap (e.g., strings, lists, etc.)
- Allocation is usually explicit (e.g., CONS in LISP, NEW in PASCAL)
- Deallocation is explicit or implicit
  1. PL/1: explicit via FREE
  2. LISP: implicit via garbage collection
     - process of determining what part of storage is no longer being referenced and returning it to the available pool
- Ex:

```
procedure reverse(x);
if empty(x) then x
else reverse2(x,Ω);

procedure reverse2(x,y)
if empty(x) then y
else reverse2(rest(x),add(first(x),y));
```

- `add(x,y)` grabs a 2-element cell from AVAIL and yields | x | y |

`reverse[(A B C)]` ⇒ `reverse2[(A B C),Ω]`

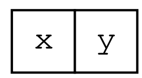


GARBAGE COLLECTION

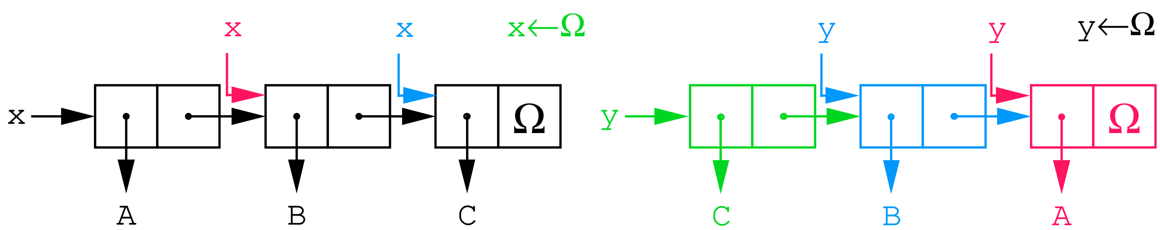
- Management of storage allocation from a heap (e.g., strings, lists, etc.)
- Allocation is usually explicit (e.g., CONS in LISP, NEW in PASCAL)
- Deallocation is explicit or implicit
  1. PL/1: explicit via FREE
  2. LISP: implicit via garbage collection
     - process of determining what part of storage is no longer being referenced and returning it to the available pool
- Ex:

```
procedure reverse(x);
if empty(x) then x
else reverse2(x,Ω);

procedure reverse2(x,y)
if empty(x) then y
else reverse2(rest(x),add(first(x),y));
```

- `add(x,y)` grabs a 2-element cell from AVAIL and yields 

`reverse[(A B C)]` ⇒ `reverse2[(A B C),Ω]`

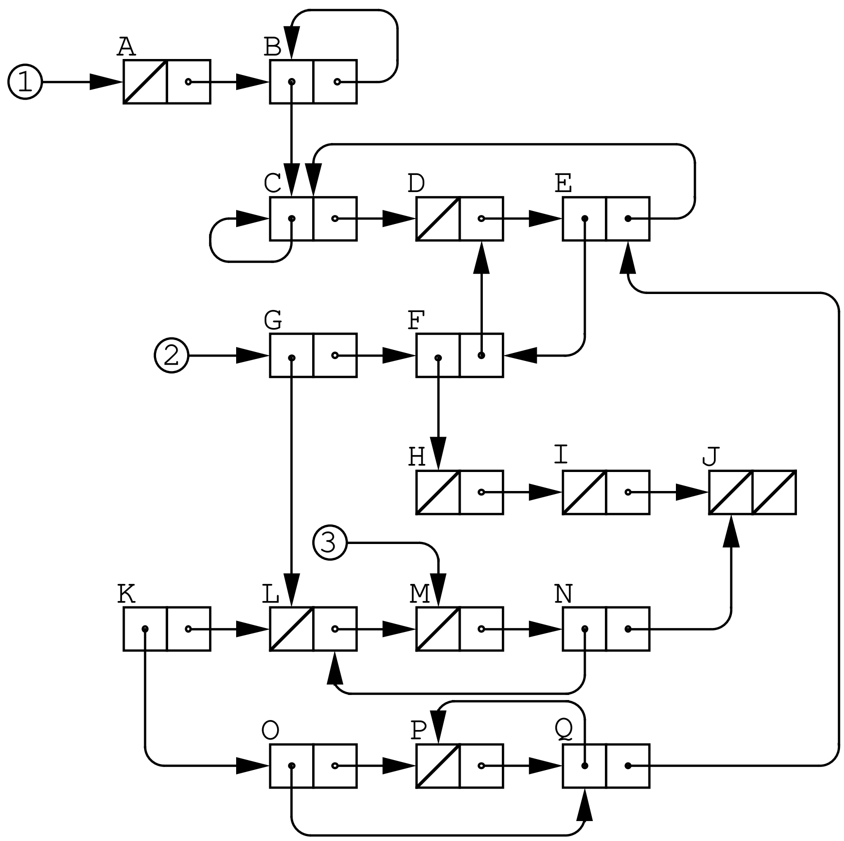


- 3 new cells have been allocated
- List `(A B C)` previously pointed at by `x` is no longer relevant or pointed at by the result of `reverse`
- `(A B C)` is garbage if no other part of the program can access it

STORAGE MAP

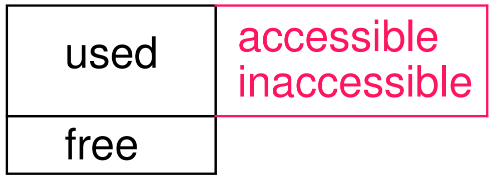
- View of memory

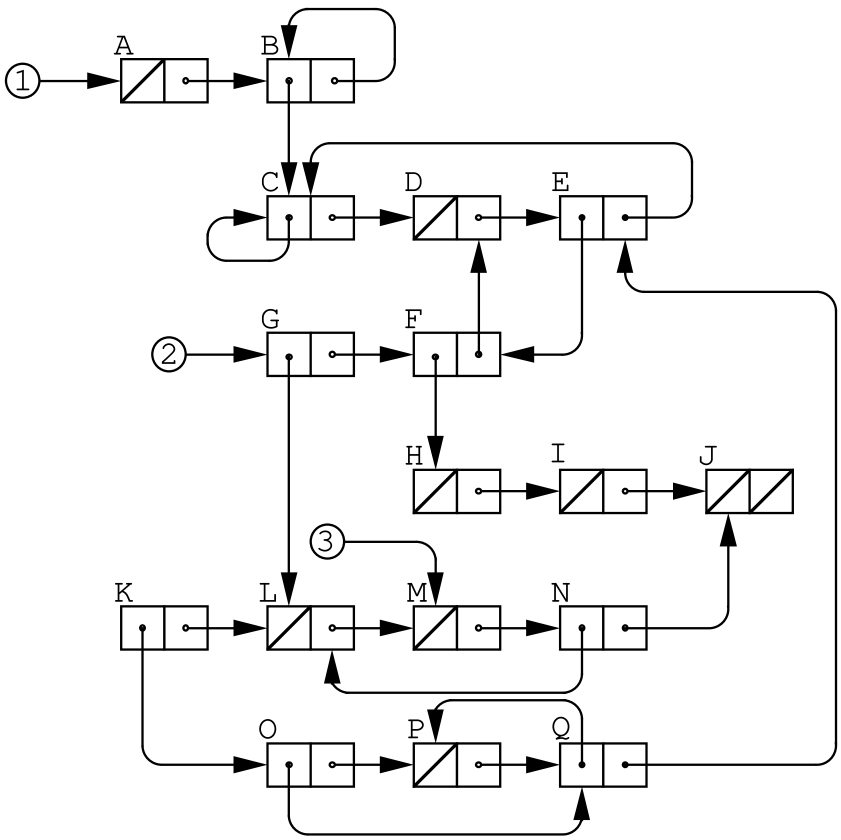
used
free



STORAGE MAP

- View of memory

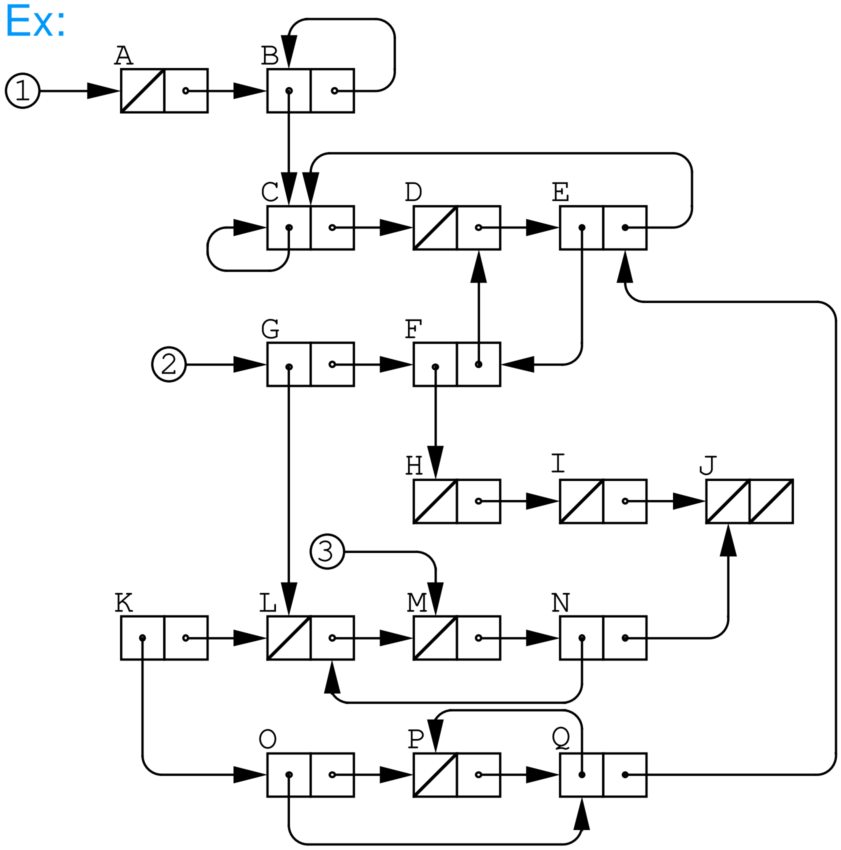




STORAGE MAP

- View of memory

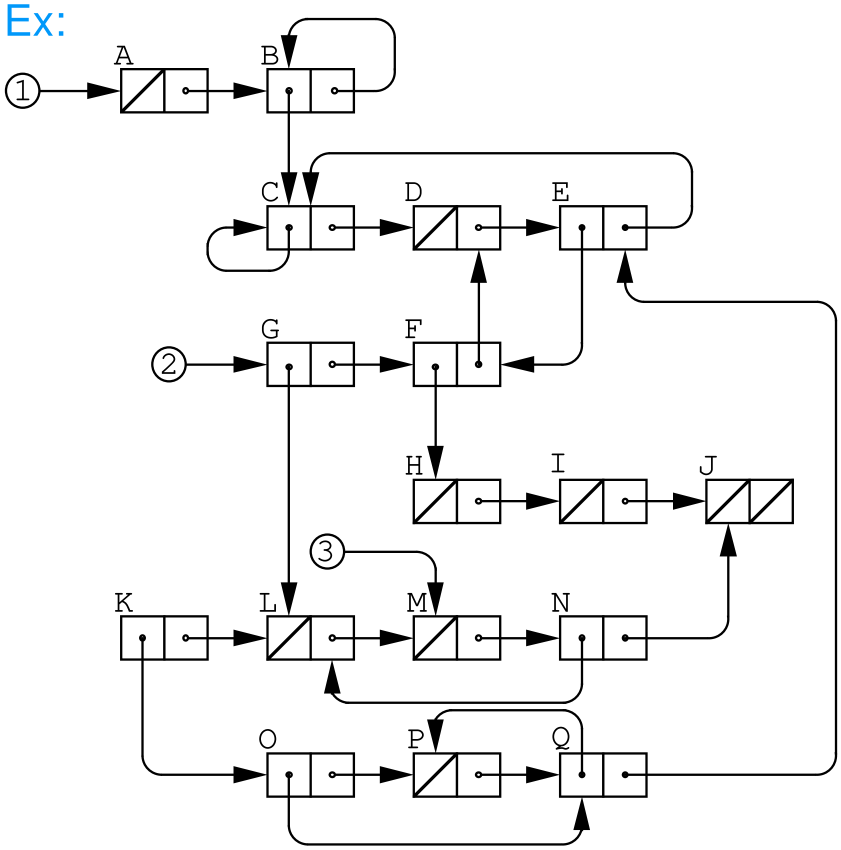
- Garbage collection is the process of coalescing FREE and INACCESSIBLE to form a new FREE area
- Ex:



STORAGE MAP

- View of memory

- Garbage collection is the process of coalescing FREE and INACCESSIBLE to form a new FREE area
- Ex:

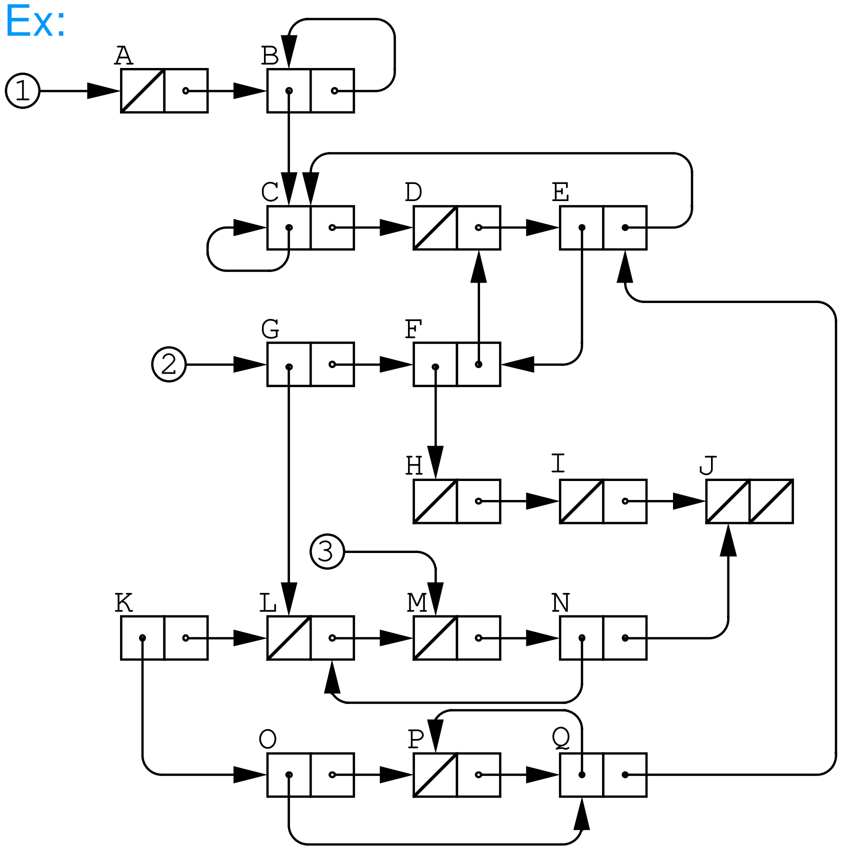


1. A, G, and M are the immediately accessible cells
   - mark all accessible cells
   - A, B, C, D, E, F, G, H, I, J, L, M, and N

STORAGE MAP

- View of memory

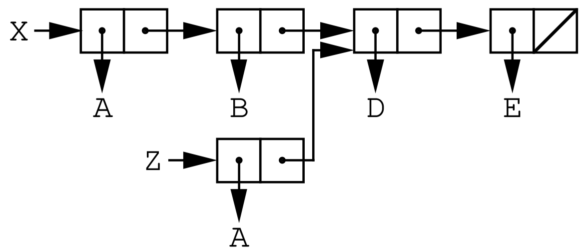
- Garbage collection is the process of coalescing FREE and INACCESSIBLE to form a new FREE area
- Ex:



1. `A`, `G`, and `M` are the immediately accessible cells
   - mark all accessible cells
   - `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, `I`, `J`, `L`, `M`, and `N`
2. `K`, `O`, `P`, and `Q` are inaccessible and are garbage collected

STORAGE MANAGEMENT TECHNIQUES

1. User keeps direct control
   - danger is dangling references
   - if storage shared, freeing a block causes problems if another part of program believes the block is still in use
   - Ex:



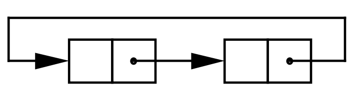
   a. `'(D E)` is shared by the binding of `X` and `Z`

   b. deallocating binding of `Z` leaves a dangling reference to `'(D E)` in the binding of `X`

2. System control via use of reference counters
   - increment whenever allocate storage or a cell is bound to an additional variable
   - decrement when cell is no longer being referenced (i.e., at block exit)
   - deallocation when reference counter is zero
   - drawbacks

   a. counters require space
      - can store separately from data (separation of data and control)

   b. recursive structures such as circular lists will never have their reference counters go to zero



3. Garbage collection
   - system replenishes available pool of storage

MECHANICS OF GARBAGE COLLECTION

- Two phases:
  1. mark all accessible nodes
  2. sweep through memory placing all unmarked nodes in available pool of storage
- Problems:
  1. slow when memory is almost full
  2. invalid information in pointer fields
  3. mark bits take up space
     - can store in a separate table
  4. limited amount of memory for garbage collection process to run

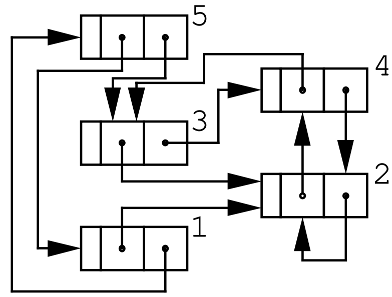
ALGORITHM 1

- Sequentially step through memory and each time encounter a marked node *p*, reset the current node to be the minimum of the unmarked addresses in the CAR and CDR fields of *p*
- Stop when reach the end of memory

```
procedure ALG1(low,high);
begin
  value integer low,high;
  global address array m;
  integer i,j;
  i←low;
  while i≤high do
    begin
      if ATOM(m[i]) or NOT(MARK(m[i])) then i←i+1
      else
        begin
          j←i+1;
          if NOT(MARK(CAR(m[i]))) then
            begin
              MARK(CAR(m[i]))←true;
              if NOT(ATOM(CAR(m[i]))) then
                  j←min(j,CAR(m[i]));
            end;
          if NOT(MARK(CDR(m[i]))) then
            begin
              MARK(CDR(m[i]))←true;
              if NOT(ATOM(CDR(m[i]))) then
                  j←min(j,CDR(m[i]));
            end;
          i←j;
        end;
    end;
end;
```

- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	0	2	4	3
2	1	4	2	2
1	0	2	5	1



1. only 2 and 4 initially accessible; start sequential scan at 1
2. 2 and its CAR and CDR already marked; resume scan at 3

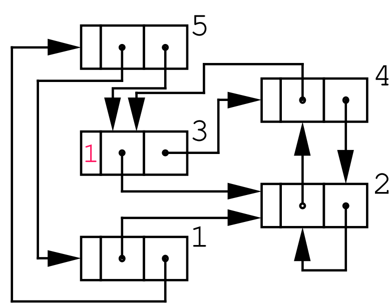
ALGORITHM 1

- Sequentially step through memory and each time encounter a marked node *p*, reset the current node to be the minimum of the unmarked addresses in the CAR and CDR fields of *p*
- Stop when reach the end of memory

```
procedure ALG1(low,high);
begin
  value integer low,high;
  global address array m;
  integer i,j;
  i←low;
  while i≤high do
    begin
      if ATOM(m[i]) or NOT(MARK(m[i])) then i←i+1
      else
        begin
          j←i+1;
          if NOT(MARK(CAR(m[i]))) then
            begin
              MARK(CAR(m[i]))←true;
              if NOT(ATOM(CAR(m[i]))) then
                  j←min(j,CAR(m[i]));
            end;
          if NOT(MARK(CDR(m[i]))) then
            begin
              MARK(CDR(m[i]))←true;
              if NOT(ATOM(CDR(m[i]))) then
                  j←min(j,CDR(m[i]));
            end;
          i←j;
        end;
    end;
end;
```

- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	0	2	4	3
2	1	4	2	2
1	0	2	5	1



1. only 2 and 4 initially accessible; start sequential scan at 1
2. 2 and its CAR and CDR already marked; resume scan at 3
3. 4 and CDR(4) are marked while CAR(4) is not marked; resume scan at 3 after marking it

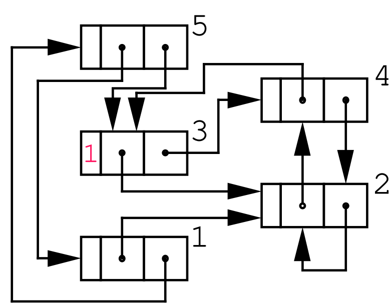
ALGORITHM 1

- Sequentially step through memory and each time encounter a marked node *p*, reset the current node to be the minimum of the unmarked addresses in the CAR and CDR fields of *p*
- Stop when reach the end of memory

```
procedure ALG1(low,high);
begin
  value integer low,high;
  global address array m;
  integer i,j;
  i←low;
  while i≤high do
    begin
      if ATOM(m[i]) or NOT(MARK(m[i])) then i←i+1
      else
        begin
          j←i+1;
          if NOT(MARK(CAR(m[i]))) then
            begin
              MARK(CAR(m[i]))←true;
              if NOT(ATOM(CAR(m[i]))) then
                  j←min(j,CAR(m[i]));
            end;
          if NOT(MARK(CDR(m[i]))) then
            begin
              MARK(CDR(m[i]))←true;
              if NOT(ATOM(CDR(m[i]))) then
                  j←min(j,CDR(m[i]));
            end;
          i←j;
        end;
    end;
end;
```

- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	0	2	4	3
2	1	4	2	2
1	0	2	5	1



1. only 2 and 4 initially accessible; start sequential scan at 1
2. 2 and its CAR and CDR already marked; resume scan at 3
3. 4 and CDR(4) are marked while CAR(4) is not marked; resume scan at 3 after marking it
4. CAR(3) and CDR(3) are already marked; resume scan at 4

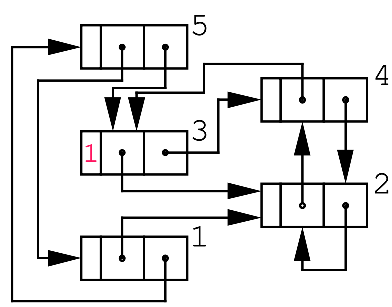
ALGORITHM 1

- Sequentially step through memory and each time encounter a marked node *p*, reset the current node to be the minimum of the unmarked addresses in the CAR and CDR fields of *p*
- Stop when reach the end of memory

```
procedure ALG1(low,high);
begin
  value integer low,high;
  global address array m;
  integer i,j;
  i←low;
  while i≤high do
    begin
      if ATOM(m[i]) or NOT(MARK(m[i])) then i←i+1
      else
        begin
          j←i+1;
          if NOT(MARK(CAR(m[i]))) then
            begin
              MARK(CAR(m[i]))←true;
              if NOT(ATOM(CAR(m[i]))) then
                  j←min(j,CAR(m[i]));
            end;
          if NOT(MARK(CDR(m[i]))) then
            begin
              MARK(CDR(m[i]))←true;
              if NOT(ATOM(CDR(m[i]))) then
                  j←min(j,CDR(m[i]));
            end;
          i←j;
        end;
    end;
end;
```

- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	0	2	4	3
2	1	4	2	2
1	0	2	5	1



1. only 2 and 4 initially accessible; start sequential scan at 1
2. 2 and its CAR and CDR already marked; resume scan at 3
3. 4 and CDR(4) are marked while CAR(4) is not marked; resume scan at 3 after marking it
4. CAR(3) and CDR(3) are already marked; resume scan at 4
5. 4 and its CAR and CDR fields are already marked; resume scan at 5 which is unmarked; exit

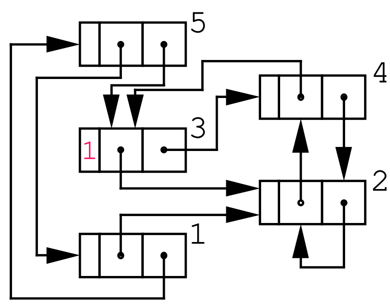
ALGORITHM 1

- Sequentially step through memory and each time encounter a marked node *p*, reset the current node to be the minimum of the unmarked addresses in the CAR and CDR fields of *p*
- Stop when reach the end of memory

```
procedure ALG1(low,high);
begin
  value integer low,high;
  global address array m;
  integer i,j;
  i←low;
  while i≤high do
    begin
      if ATOM(m[i]) or NOT(MARK(m[i])) then i←i+1
      else
        begin
          j←i+1;
          if NOT(MARK(CAR(m[i]))) then
            begin
              MARK(CAR(m[i]))←true;
              if NOT(ATOM(CAR(m[i]))) then
                  j←min(j,CAR(m[i]));
            end;
          if NOT(MARK(CDR(m[i]))) then
            begin
              MARK(CDR(m[i]))←true;
              if NOT(ATOM(CDR(m[i]))) then
                  j←min(j,CDR(m[i]));
            end;
          i←j;
        end;
    end;
end;
```

- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	0	2	4	3
2	1	4	2	2
1	0	2	5	1



1. only 2 and 4 initially accessible; start sequential scan at 1
2. 2 and its CAR and CDR already marked; resume scan at 3
3. 4 and CDR(4) are marked while CAR(4) is not marked; resume scan at 3 after marking it
4. CAR(3) and CDR(3) are already marked; resume scan at 4
5. 4 and its CAR and CDR fields are already marked; resume scan at 5 which is unmarked; exit

- Algorithm is slow – $O(m \times n)$ for m cells in memory and n marked cells

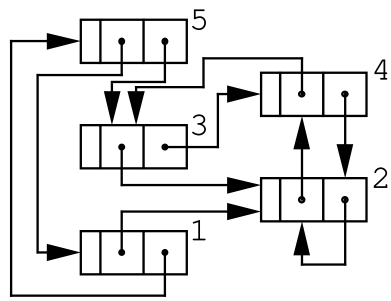
ALGORITHM 2

- Use a stack to keep track of cells to be marked
  1. mark all immediately accessible cells and enter them in stack
  2. remove entries from stack and for each one that is not an atom, mark its CAR and CDR fields and enter them on the stack if they have not been previously marked
  3. terminate when the stack is empty
- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	0	2	4	3
2	1	4	2	2
1	0	2	5	1

stack:

4
2



  1. initially, only 2 and 4 are accessible and entered on the stack

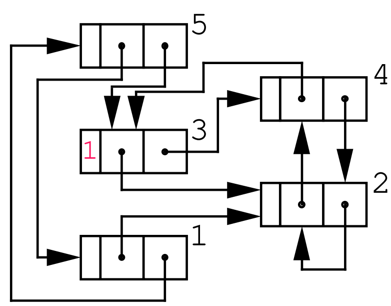
ALGORITHM 2

- Use a stack to keep track of cells to be marked
  1. mark all immediately accessible cells and enter them in stack
  2. remove entries from stack and for each one that is not an atom, mark its CAR and CDR fields and enter them on the stack if they have not been previously marked
  3. terminate when the stack is empty
- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	~~0~~ 1	2	4	3
2	1	4	2	2
1	0	2	5	1

stack:

4	3
2	2



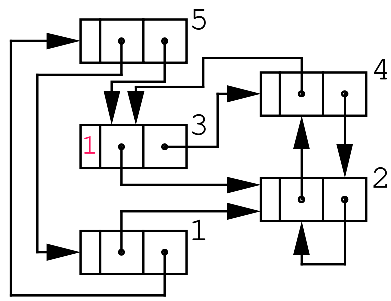
  1. initially, only 2 and 4 are accessible and entered on the stack
  2. remove 4 from the stack; enter CAR(4)=3 on the stack and mark it

ALGORITHM 2

- Use a stack to keep track of cells to be marked
  1. mark all immediately accessible cells and enter them in stack
  2. remove entries from stack and for each one that is not an atom, mark its CAR and CDR fields and enter them on the stack if they have not been previously marked
  3. terminate when the stack is empty
- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	~~0~~ 1	2	4	3
2	1	4	2	2
1	0	2	5	1

stack: | 4 2 | 3 2 | 2



1. initially, only 2 and 4 are accessible and entered on the stack
2. remove 4 from the stack; enter CAR(4)=3 on the stack and mark it
3. remove 3 from the stack; both CAR(3) and CDR(3) have already been marked

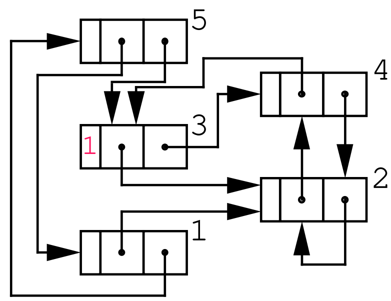
ALGORITHM 2

- Use a stack to keep track of cells to be marked
  1. mark all immediately accessible cells and enter them in stack
  2. remove entries from stack and for each one that is not an atom, mark its CAR and CDR fields and enter them on the stack if they have not been previously marked
  3. terminate when the stack is empty
- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	~~0~~1	2	4	3
2	1	4	2	2
1	0	2	5	1

stack:

4	3		
2	2	2	–



1. initially, only 2 and 4 are accessible and entered on the stack
2. remove 4 from the stack; enter CAR(4)=3 on the stack and mark it
3. remove 3 from the stack; both CAR(3) and CDR(3) have already been marked
4. remove 2 from the stack; both CAR(2) and CDR(2) have already been marked; stack is empty and finished

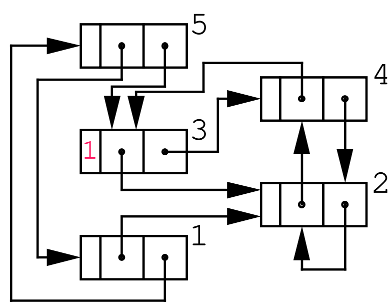
ALGORITHM 2

- Use a stack to keep track of cells to be marked
  1. mark all immediately accessible cells and enter them in stack
  2. remove entries from stack and for each one that is not an atom, mark its CAR and CDR fields and enter them on the stack if they have not been previously marked
  3. terminate when the stack is empty
- Ex:

	mark	CAR	CDR	
5	0	1	3	5
4	1	3	2	4
3	~~0~~1	2	4	3
2	1	4	2	2
1	0	2	5	1

stack:

4	3		
2	2	2	–

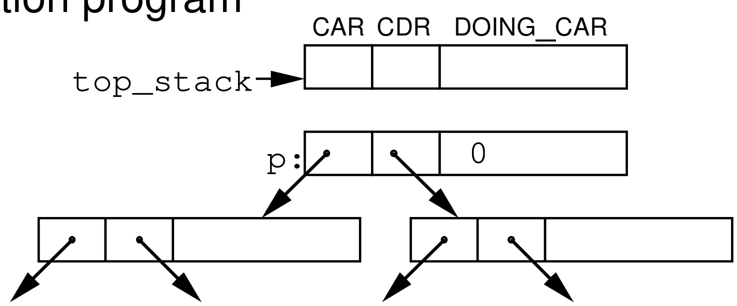


  1. initially, only 2 and 4 are accessible and entered on the stack
  2. remove 4 from the stack; enter CAR(4)=3 on the stack and mark it
  3. remove 3 from the stack; both CAR(3) and CDR(3) have already been marked
  4. remove 2 from the stack; both CAR(2) and CDR(2) have already been marked; stack is empty and finished

- Time proportional to number of cells that are marked
- Drawbacks
  1. where to store the stack?
     - little memory available at time garbage collection is invoked
  2. not inconceivable to run out of stack space when running algorithm 2

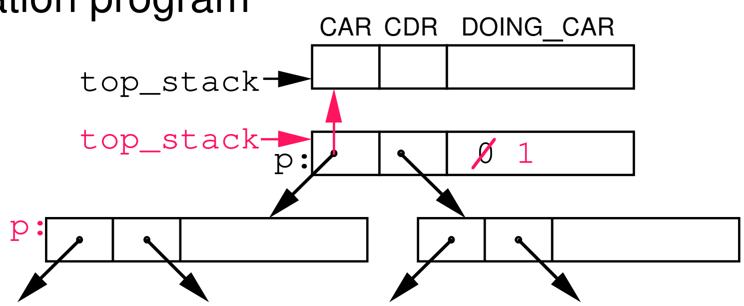
ALGORITHM 3

- Avoids use of stack in Algorithm 2 by reusing the CAR and CDR fields to simulate the stack
- Requires additional one-bit DOING_CAR field for each non-atomic cell *p* to indicate whether the cell currently being processed is pointed at by CAR(p) or CDR(p)
- DOING_CAR is useful in detecting if both CAR and CDR of *p* have been garbage-collected
- Disadvantages: cannot execute in parallel with application program
- Ex:



ALGORITHM 3

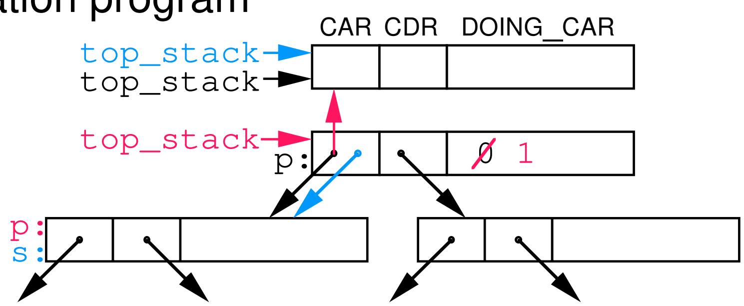
- Avoids use of stack in Algorithm 2 by reusing the CAR and CDR fields to simulate the stack
- Requires additional one-bit DOING_CAR field for each non-atomic cell *p* to indicate whether the cell currently being processed is pointed at by CAR(p) or CDR(p)
- DOING_CAR is useful in detecting if both CAR and CDR of *p* have been garbage-collected
- Disadvantages: cannot execute in parallel with application program
- Ex:



1. visiting *p* for first time (assume CAR(p) is not marked)
   - set DOING_CAR to true
   - set CAR(p) to predecessor (TOP_STACK)
   - set TOP_STACK to *p*
   - set *p* to old value of CAR(p)
   - continue processing *p*

ALGORITHM 3

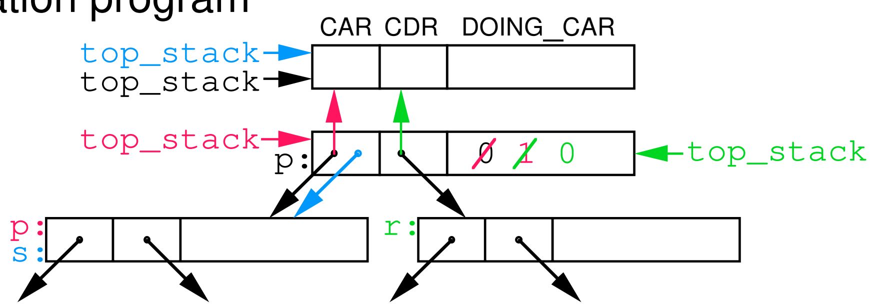
- Avoids use of stack in Algorithm 2 by reusing the CAR and CDR fields to simulate the stack
- Requires additional one-bit DOING_CAR field for each non-atomic cell *p* to indicate whether the cell currently being processed is pointed at by CAR(p) or CDR(p)
- DOING_CAR is useful in detecting if both CAR and CDR of *p* have been garbage-collected
- Disadvantages: cannot execute in parallel with application program
- Ex:



1. visiting *p* for first time (assume CAR(p) is not marked)
   - set DOING_CAR to true
   - set CAR(p) to predecessor (TOP_STACK)
   - set TOP_STACK to *p*
   - set *p* to old value of CAR(p)
   - continue processing *p*
2. visiting *p* for second time (returning from *s*)
   - reset CAR(p) to *s*
   - set TOP_STACK to previous value of CAR(p)

ALGORITHM 3

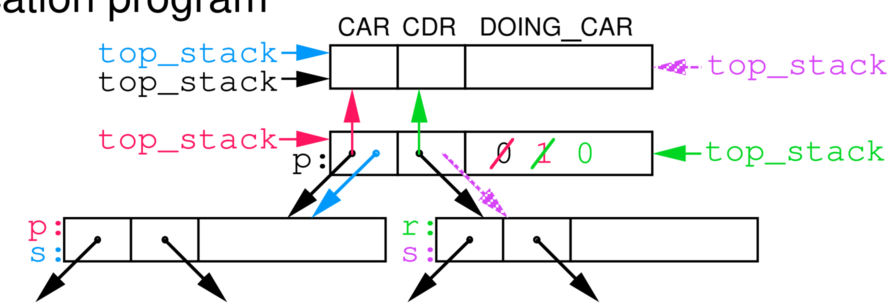
- Avoids use of stack in Algorithm 2 by reusing the CAR and CDR fields to simulate the stack
- Requires additional one-bit DOING_CAR field for each non-atomic cell *p* to indicate whether the cell currently being processed is pointed at by CAR(p) or CDR(p)
- DOING_CAR is useful in detecting if both CAR and CDR of *p* have been garbage-collected
- Disadvantages: cannot execute in parallel with application program
- Ex:



1. visiting *p* for first time (assume CAR(p) is not marked)
   - set DOING_CAR to true
   - set CAR(p) to predecessor (TOP_STACK)
   - set TOP_STACK to *p*
   - set *p* to old value of CAR(p)
   - continue processing *p*
2. visiting *p* for second time (returning from *s*)
   - reset CAR(p) to *s*
   - set TOP_STACK to previous value of CAR(p)
   - if *r*=CDR(p) is not marked then
     a. mark *r*
     b. set CDR(p) to previous TOP_STACK value
     c. set TOP_STACK to *p*
     d. set DOING_CAR to false
     e. continue processing *r*

ALGORITHM 3

- Avoids use of stack in Algorithm 2 by reusing the CAR and CDR fields to simulate the stack
- Requires additional one-bit DOING_CAR field for each non-atomic cell *p* to indicate whether the cell currently being processed is pointed at by CAR(p) or CDR(p)
- DOING_CAR is useful in detecting if both CAR and CDR of *p* have been garbage-collected
- Disadvantages: cannot execute in parallel with application program
- Ex:



1. visiting *p* for first time (assume CAR(p) is not marked)
   - set DOING_CAR to true
   - set CAR(p) to predecessor (TOP_STACK)
   - set TOP_STACK to *p*
   - set *p* to old value of CAR(p)
   - continue processing *p*
2. visiting *p* for second time (returning from *s*)
   - reset CAR(p) to *s*
   - set TOP_STACK to previous value of CAR(p)
   - if *r*=CDR(p) is not marked then
     a. mark *r*
     b. set CDR(p) to previous TOP_STACK value
     c. set TOP_STACK to *p*
     d. set DOING_CAR to false
     e. continue processing *r*
3. visiting *p* for third time (returning from *s*)
   - reset CDR(p) to *s*
   - set TOP_STACK to previous value of CDR(p)
   - continue unwinding

STOP-AND-COPY GARBAGE COLLECTION

- Problems with sweep phase of mark-and-sweep paradigm:
  1. must examine each memory location to see if marked
  2. inefficient if most locations were marked as will need to reinvoke process soon
     - probably preferable to increase size of FREE area
  3. also inefficient if few are marked (i.e., most are inaccessible)
     - may be cheaper to copy contents of marked ones while marking in which case remaining storage is available for allocation
- Alternative is to use *stop-and-copy* paradigm
  1. motivation is that most locations are inaccessible
     - cheaper to copy accessible while marking
     - instead of marking followed by a sweep through all of memory which includes the inaccessible as well
  2. partition memory into 2 areas: copied-from and copied-to
     - only one area is ever in use
  3. algorithm:
     - when storage is exhausted in the area in-use (copied-from) identify the accessible locations in it and in the process copy them to the other area (copied-to)
     - continue execution with all subsequent allocation from the unoccupied locations in copied-to
     - copied-from will serve as the copied-to area the next time storage is exhausted
     - process continues with roles of two areas interchanged
  4. drawbacks:
     - need to be careful so pointers are correct if sharing is permitted
     - one half of memory is not in use

GENERATIONAL SCAVENGING

- In terms of objects rather than locations
- Focus on age (time since initial allocation of object)
- Basic premises (backed by empirical observations)
  1. young objects die at a faster rate than old objects
  2. old objects usually refer to old objects
- Implications:
  1. storage reclamation should focus on young objects and not waste time on old objects
  2. since so many young objects die, it is cheaper to copy the surviving ones than to sweep through the corpses
- Two types of garbage collection
  1. variant of stop-and-copy (scavenging) very often
  2. mark-and-sweep less often
- Memory partitioned into four areas (termed *spaces*):
  1. NewSpace: new objects allocated from here
  2. PastSurvivorSpace: like copied-from
  3. FutureSurvivorSpace: like copied-to
  4. OldSpace: survivied more than *s* scavenges
- Terminology rationale:
  1. scavenge: only examine most promising part of storage
  2. generation: don't bother to scavenge when grown up (tenured)

MECHANICS OF GENERATIONAL SCAVENGING

- Start with immediately accessible objects
- Scavenge at regular intervals or when NewSpace is exhausted
  1. only “mark” accessible objects in NewSpace and PastSurvivorSpace and copy to FutureSurvivorSpace
     - apply recursively
  2. tenure objects (move into OldSpace) if they have survived more than *s* scavenges (moves from PastSurvivorSpace to FutureSurvivorSpace)
  3. do not pursue pointers to objects in OldSpace
- Inaccessible old objects are collected by applying mark-and-sweep algorithm to entire memory
  1. OldSpace
  2. YoungSpace = {NewSpace,PastSurvivorSpace, FutureSurvivorSpace}
- Remembered set contains objects in OldSpace that point to objects in YoungSpace
- Time complexity: number of surviving objects (including those accessible from the remembered set)
- Space complexity:
  1. 1 bit per object to indicate if moved from PastSurvivorSpace to FutureSurvivorSpace
  2. counters for tenuring

MECHANICS OF REMEMBERED SET *R*

- Objects added when:
  1. assignment operation results in an object in OldSpace pointing to an object in YoungSpace (e.g., RPLACA and RPLACD in LISP)
  2. object being tenured points to an object in YoungSpace
- All objects *p* pointed at by object *o* in *R* are scavenged:
  1. if *p* points to OldSpace, then leave *p* alone
  2. if *p* references YoungSpace, then:
     - if *p* already moved to *f* in FutureSurvivorSpace or OldSpace, then update pointer in *o* to point to *f* instead of to *p*
     - if *p* has not been moved already, then:
       a. move *p* to *f* in either FutureSurvivorSpace or OldSpace, depending on *p*'s age
       b. replace contents of *p* by a pointer to *f*
       c. update pointer in *o* to refer to *f* instead of *p*
  3. if after scavenge update, all objects *p* pointed by *o* are in OldSpace, then remove *o* from *R*
- Recursively process (scavenge) all objects moved to FutureSurvivorSpace, OldSpace, and the remembered set