DYNAMIC STORAGE ALLOCATION

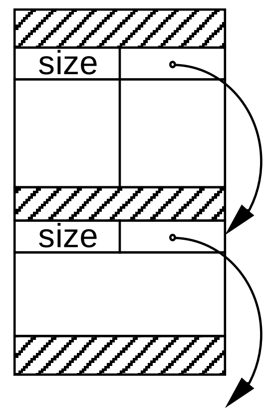
Hanan Samet

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

DYNAMIC STORAGE ALLOCATION

- Explicit allocation and deallocation ('freeing' or 'liberating') of blocks of contiguous storage locations
- Issues:

1. how to keep track of available space and its partitioning
   - usually keep a linked list of available blocks
     a. elements
        - location of start of block
        - size of block
        - pointer to next block in list
     b. how to order (i.e., 'sort') list
        - by location (i.e., increasing order)
        - by size
        - no order



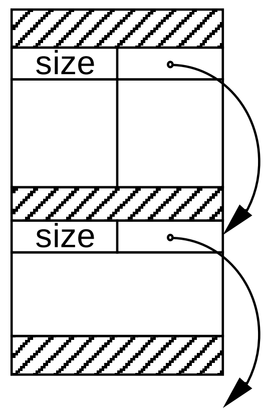
2. how to find a block of *b* consecutive locations
   - if list sorted by location, find first one with $s \geq b$ (*first fit*)
     a. requires a search
     b. but good if want to merge adjacent empty blocks into larger ones upon storage deallocation
   - if list sorted by size, find smallest one with $s \geq b$ (*best fit*)

- Ex: first fit is superior to best fit

request	available areas first fit	available areas best fit
start	1300,1200	1300,1200

DYNAMIC STORAGE ALLOCATION

- Explicit allocation and deallocation ('freeing' or 'liberating') of blocks of contiguous storage locations
- Issues:
  1. how to keep track of available space and its partitioning
     - usually keep a linked list of available blocks
       a. elements
          - location of start of block
          - size of block
          - pointer to next block in list
       b. how to order (i.e., 'sort') list
          - by location (i.e., increasing order)
          - by size
          - no order



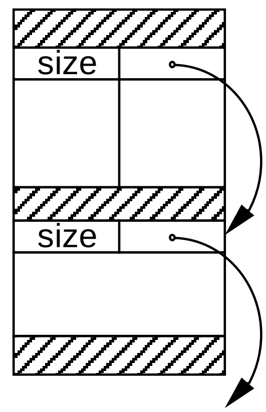
  2. how to find a block of *b* consecutive locations
     - if list sorted by location, find first one with $s \geq b$ (*first fit*)
       a. requires a search
       b. but good if want to merge adjacent empty blocks into larger ones upon storage deallocation
     - if list sorted by size, find smallest one with $s \geq b$ (*best fit*)
- Ex: first fit is superior to best fit

request	available areas first fit	available areas best fit
start	1300,1200	1300,1200
1000	300,1200	1300,200

DYNAMIC STORAGE ALLOCATION

- Explicit allocation and deallocation ('freeing' or 'liberating') of blocks of contiguous storage locations
- Issues:
  1. how to keep track of available space and its partitioning
     - usually keep a linked list of available blocks
       a. elements
          - location of start of block
          - size of block
          - pointer to next block in list

       b. how to order (i.e., 'sort') list
          - by location (i.e., increasing order)
          - by size
          - no order

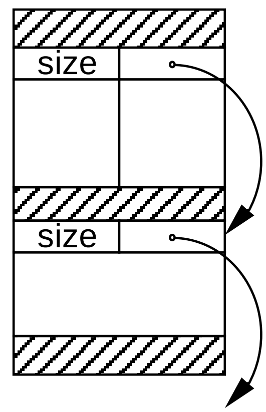


  2. how to find a block of *b* consecutive locations
     - if list sorted by location, find first one with $s \geq b$ (*first fit*)
       a. requires a search
       b. but good if want to merge adjacent empty blocks into larger ones upon storage deallocation
     - if list sorted by size, find smallest one with $s \geq b$ (*best fit*)
- Ex: first fit is superior to best fit

request	available areas first fit	available areas best fit
start	1300,1200	1300,1200
1000	300,1200	1300,200
1100	300,100	200,200

DYNAMIC STORAGE ALLOCATION

- Explicit allocation and deallocation (‘freeing’ or ‘liberating’) of blocks of contiguous storage locations
- Issues:
  1. how to keep track of available space and its partitioning
     - usually keep a linked list of available blocks
       a. elements
          - location of start of block
          - size of block
          - pointer to next block in list
       b. how to order (i.e., ‘sort’) list
          - by location (i.e., increasing order)
          - by size
          - no order

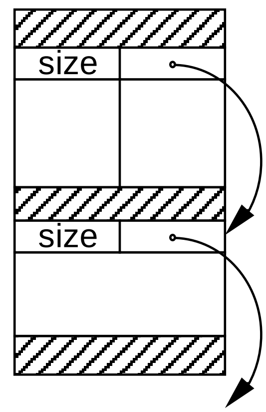


  2. how to find a block of *b* consecutive locations
     - if list sorted by location, find first one with $s \geq b$ (*first fit*)
       a. requires a search
       b. but good if want to merge adjacent empty blocks into larger ones upon storage deallocation
     - if list sorted by size, find smallest one with $s \geq b$ (*best fit*)
- Ex: first fit is superior to best fit

request	available areas first fit	available areas best fit
start	1300,1200	1300,1200
1000	300,1200	1300,200
1100	300,100	200,200
250	50,100	STUCK!

DYNAMIC STORAGE ALLOCATION

- Explicit allocation and deallocation ('freeing' or 'liberating') of blocks of contiguous storage locations
- Issues:
  1. how to keep track of available space and its partitioning
     - usually keep a linked list of available blocks
       a. elements
          - location of start of block
          - size of block
          - pointer to next block in list



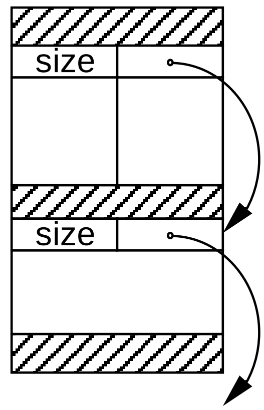
       b. how to order (i.e., 'sort') list
          - by location (i.e., increasing order)
          - by size
          - no order
  2. how to find a block of b consecutive locations
     - if list sorted by location, find first one with $s \geq b$ (*first fit*)
       a. requires a search
       b. but good if want to merge adjacent empty blocks into larger ones upon storage deallocation
     - if list sorted by size, find smallest one with $s \geq b$ (*best fit*)
- Ex: first fit is superior to best fit

request	available areas first fit	available areas best fit
start	1300,1200	1300,1200
1000	300,1200	1300,200
1100	300,100	200,200
250	50,100	STUCK!

- Requests in order of increasing size: first fit is better
- Requests in order of decreasing size: best fit is better

DYNAMIC STORAGE ALLOCATION

- Explicit allocation and deallocation ('freeing' or 'liberating') of blocks of contiguous storage locations
- Issues:
  1. how to keep track of available space and its partitioning
     - usually keep a linked list of available blocks
       a. elements
          - location of start of block
          - size of block
          - pointer to next block in list
       b. how to order (i.e., 'sort') list
          - by location (i.e., increasing order)
          - by size
          - no order



  2. how to find a block of *b* consecutive locations
     - if list sorted by location, find first one with $s \geq b$ (*first fit*)
       a. requires a search
       b. but good if want to merge adjacent empty blocks into larger ones upon storage deallocation
     - if list sorted by size, find smallest one with $s \geq b$ (*best fit*)
- Ex: first fit is superior to best fit

request	available areas first fit	available areas best fit
start	1300,1200	1300,1200
1000	300,1200	1300,200
1100	300,100	200,200
250	50,100	STUCK!

- Requests in order of increasing size: first fit is better
- Requests in order of decreasing size: best fit is better
- Can give example where best fit is better than first fit

FRAGMENTATION

- Fragmentation results when too many small blocks are generated
- Solutions:
  1. can avoid by choosing a constant k and selecting block a of size s to satisfy the request for a block of size b if $s - b < k$
     - eliminates small blocks
     - speeds up search in first-fit method as list of blocks is smaller
  2. can avoid inspecting blocks that are too small in first-fit by performing search in a circular manner so that it resumes where the last block was found
  3. can also avoid by using compaction upon deallocation

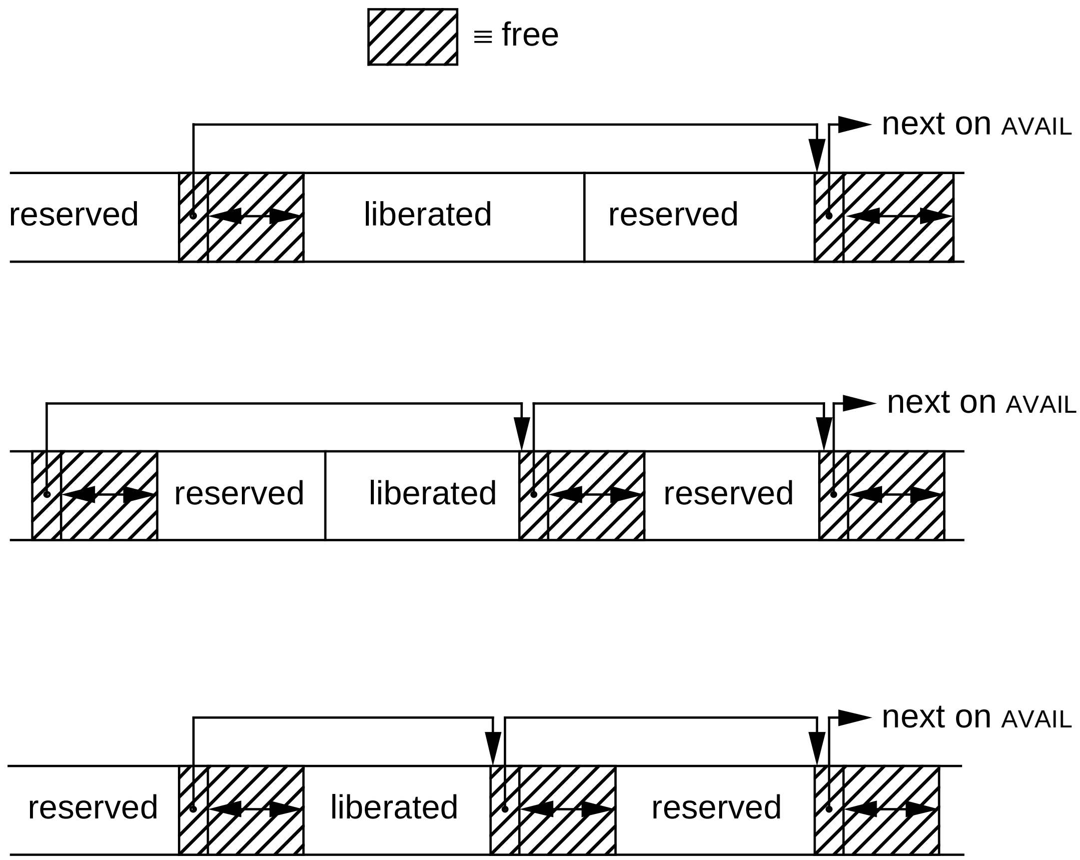
LIBERATION

1. Want to return storage to the AVAIL list as soon as possible
   - implies that can coalesce elements of AVAIL list into larger blocks
2. Contrast with methods based on garbage collection which allocate storage continuously until exhausting the AVAIL list
   - followed by a pass for storage reclamation and compaction
3. Combining garbage collection with compaction
   - storage locations must be moved
   - need to exercise care when moving pointer data
   - presence of relocation registers obviates some of the problems, since the pointers could be offset addresses

LIBERATION WITH COALESCING

Ex: assume a sorted AVAIL list by memory locations

- i.e., LINK(p)≠Ω ⇒ LINK(p)>p

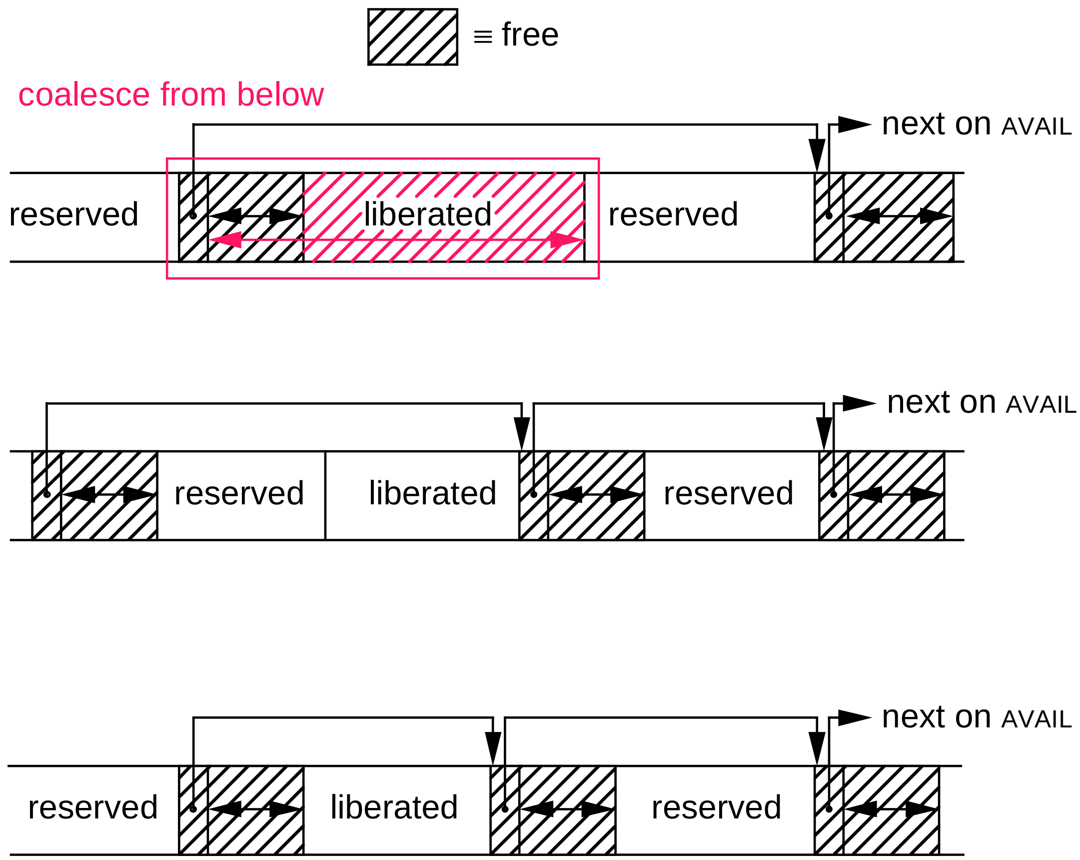


Problem: each time the algorithm is invoked to liberate block pointed at by p, we must search through approximately half the list to locate q such that LINK(q)>p

LIBERATION WITH COALESCING

Ex: assume a sorted AVAIL list by memory locations

- i.e., $\text{LINK}(p) \neq \Omega \Rightarrow \text{LINK}(p) > p$

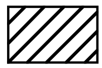


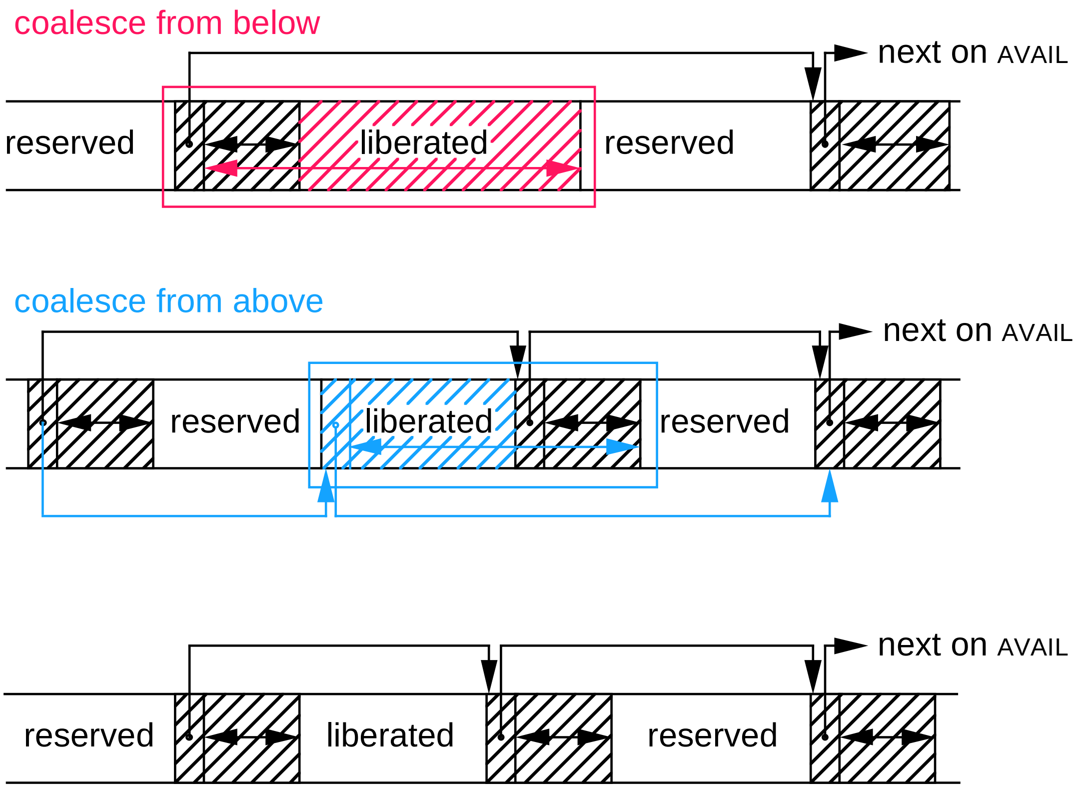
Problem: each time the algorithm is invoked to liberate block pointed at by p, we must search through approximately half the list to locate q such that LINK(q)>p

LIBERATION WITH COALESCING

Ex: assume a sorted AVAIL list by memory locations

- i.e., LINK(p)$\neq\Omega \Rightarrow$ LINK(p)>p

 ≡ free

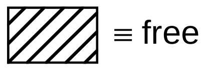


Problem: each time the algorithm is invoked to liberate block pointed at by p, we must search through approximately half the list to locate q such that LINK(q)>p

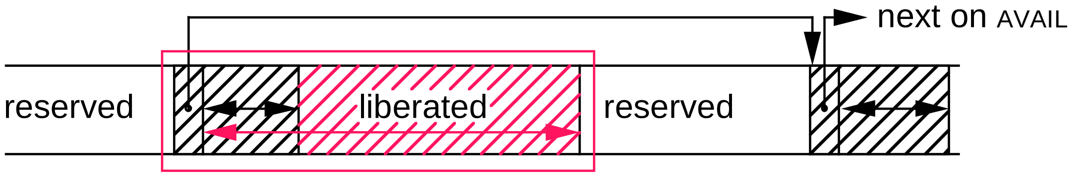
LIBERATION WITH COALESCING

Ex: assume a sorted AVAIL list by memory locations

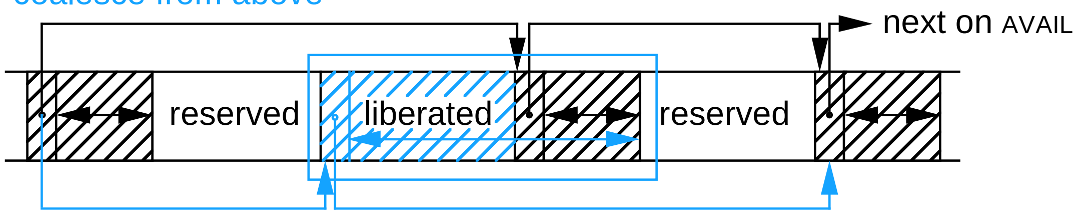
- i.e., LINK(p)≠Ω ⇒ LINK(p)>p



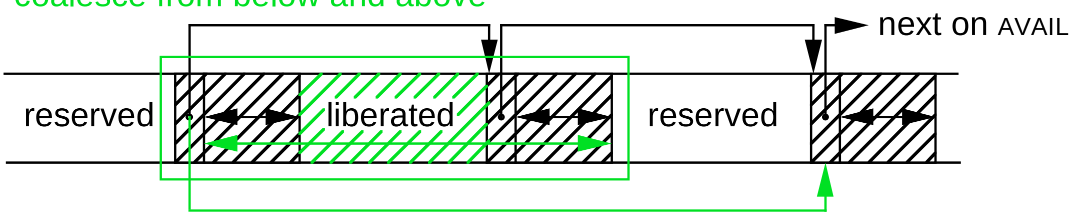
coalesce from below



coalesce from above



coalesce from below and above



Problem: each time the algorithm is invoked to liberate block pointed at by p, we must search through approximately half the list to locate q such that LINK(q)>p

LIBERATION ALGORITHM

- Assume `N` consecutive words starting at `P0` are being liberated
- Algorithm:

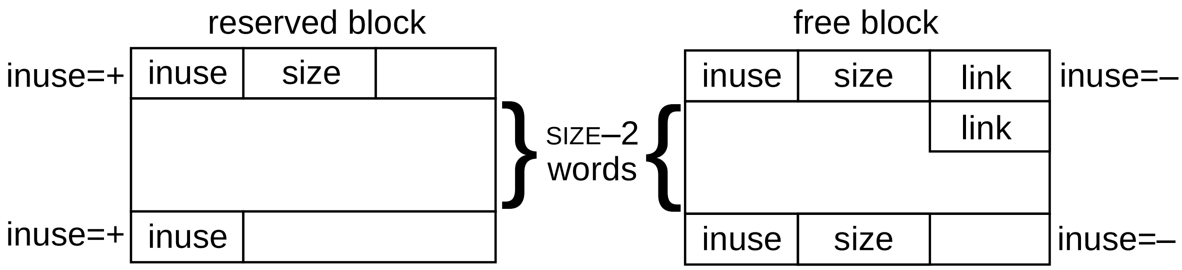
1. search through `AVAIL` until finding a node `Q` such that `link(Q) = P > P0`

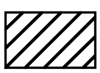
2.
```
if P0+N = P then
  begin  /* coalesce from above */
    size(P0)←size(P)+N;
    link(P0)←link(P);
  end
else
  begin
    link(P0)←P;
    size(P0)←N;
  end;
```

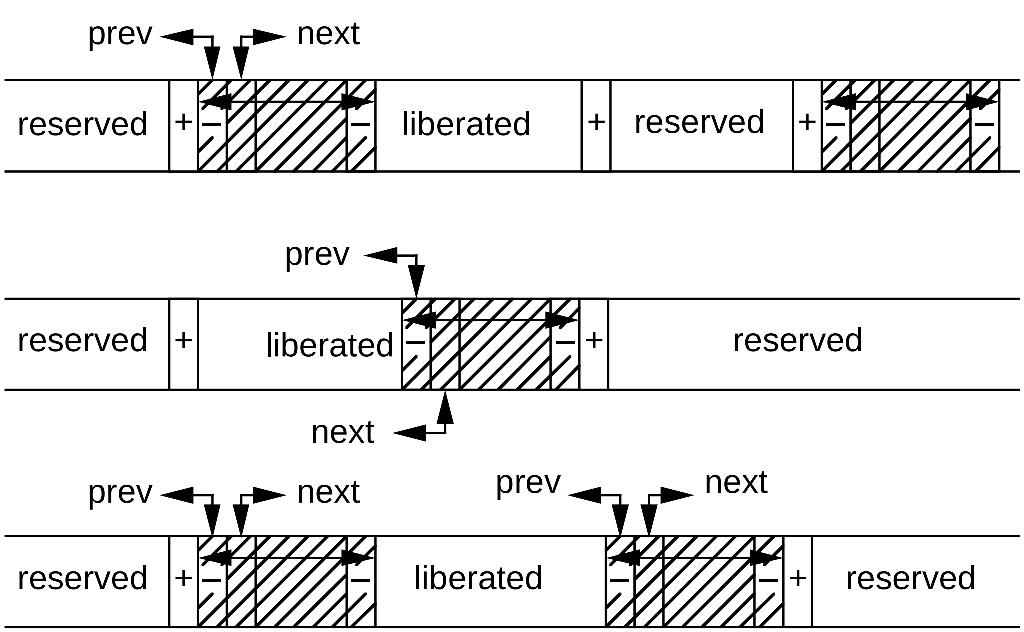
3.
```
if Q+size(Q) = P0 then
  begin  /* coalesce from below */
    size(Q)←size(Q)+size(P);
    /* N was already accounted for in step 2 (above) */
    link(Q)←link(P0);
  end
else link(Q)←P0;
```

LIBERATION USING DOUBLY-LINKED LISTS

- Data structure

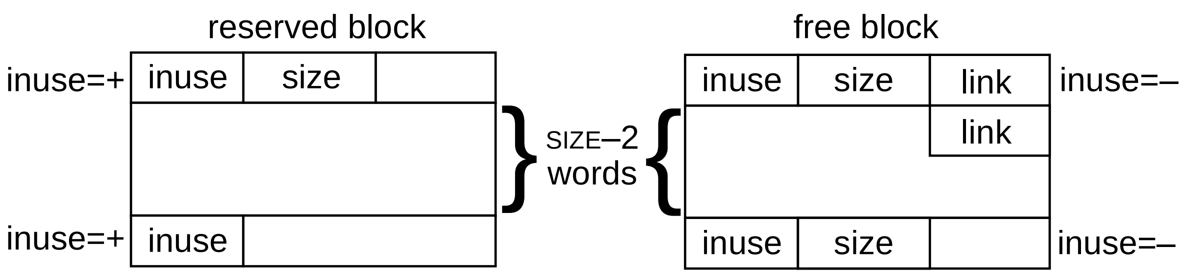


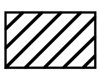
- INUSE and SIZE fields
  1. easy to locate immediately adjacent blocks to determine if coalescing is possible
  2. obviate need to sort list of available blocks (AVAIL) in increasing memory size
  3. more complex if sort AVAIL by block size as need to update
- Doubly-linked AVAIL enables easy removal of coalesced blocks
- Ex:  ≡ free

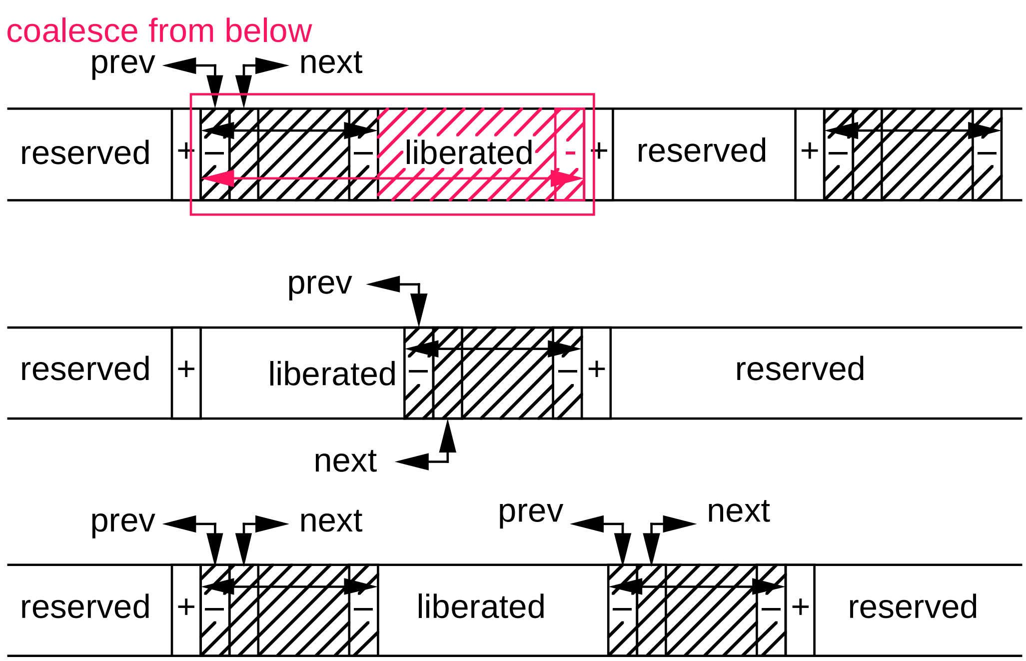


LIBERATION USING DOUBLY-LINKED LISTS

- Data structure

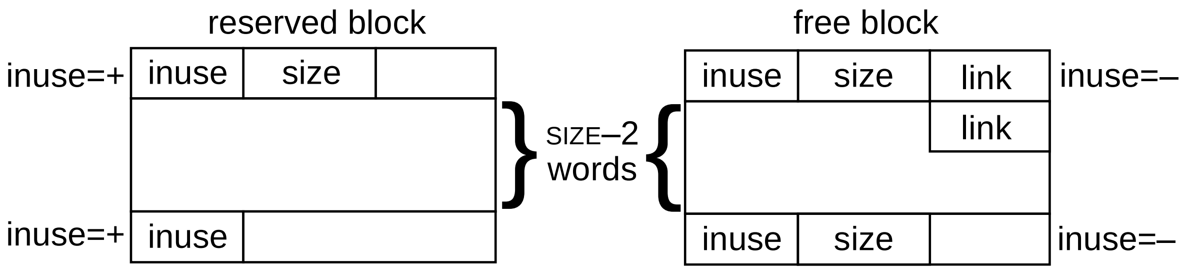


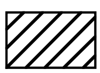
- INUSE and SIZE fields
  1. easy to locate immediately adjacent blocks to determine if coalescing is possible
  2. obviate need to sort list of available blocks (AVAIL) in increasing memory size
  3. more complex if sort AVAIL by block size as need to update
- Doubly-linked AVAIL enables easy removal of coalesced blocks
- Ex:  ≡ free

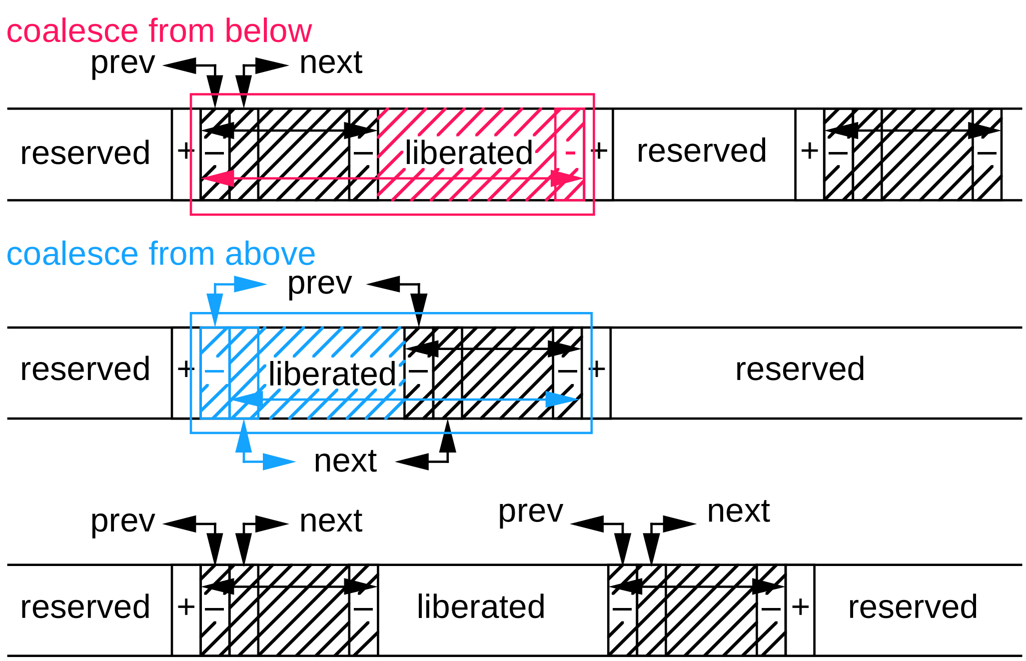


LIBERATION USING DOUBLY-LINKED LISTS

- Data structure

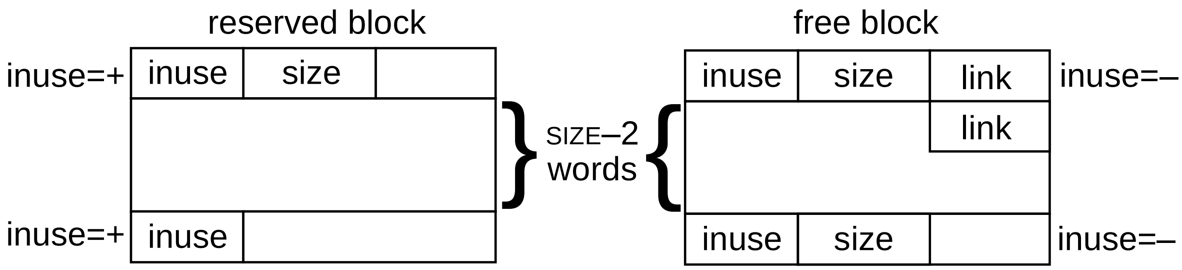


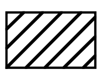
- INUSE and SIZE fields
  1. easy to locate immediately adjacent blocks to determine if coalescing is possible
  2. obviate need to sort list of available blocks (AVAIL) in increasing memory size
  3. more complex if sort AVAIL by block size as need to update
- Doubly-linked AVAIL enables easy removal of coalesced blocks
- Ex:  ≡ free

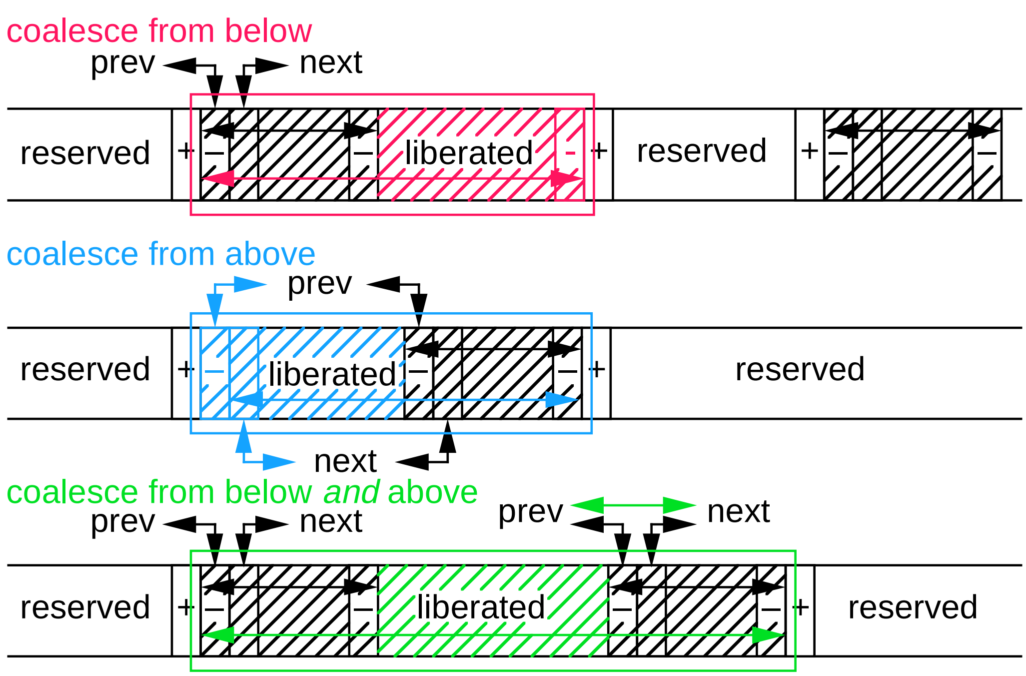


LIBERATION USING DOUBLY-LINKED LISTS

- Data structure



- INUSE and SIZE fields
  1. easy to locate immediately adjacent blocks to determine if coalescing is possible
  2. obviate need to sort list of available blocks (AVAIL) in increasing memory size
  3. more complex if sort AVAIL by block size as need to update
- Doubly-linked AVAIL enables easy removal of coalesced blocks
- Ex:  ≡ free



BUDDY SYSTEM

- Restrict block size to be a power of 2
  1. all blocks of size 2^k start at location x where $x \bmod 2^k = 0$
  2. given a block starting at location x such that $x \bmod 2^k = 0$
     - $\text{BUDDY}_k(x) = x + 2^k$ if $x \bmod 2^{k+1} = 0$
     - $\text{BUDDY}_k(x) = x - 2^k$ if $x \bmod 2^{k+1} = 2^k$
     - Ex: $\text{BUDDY}_2(10100) = 10000$
  3. only buddies can be merged
  4. try to coalesce buddies when storage is deallocated
- k different available block lists – one for each block size
- When request a block of size 2^k and none is available:
  1. split smallest block $2^j > 2^k$ into a pair of blocks of size 2^{j-1}
  2. place block on appropriate AVAIL list and try again
- Data structure
  1. doubly-linked list (not circular) FREE of available blocks indexed by k
     - links stored in actual blocks
     - FREE[k] points to first available block of size 2^k
  2. each block contains
     - INUSE bit
     - SIZE
     - NEXT and PREV links for FREE list
- Can get greater variety in block sizes using Fibonacci sequence of block sizes so $b_i = b_{i-1}+b_{i-2}$ and now ratio of successive block sizes is 2/3 instead of 1/2

EXAMPLE OF BUDDY ALGORITHM

- M = 4

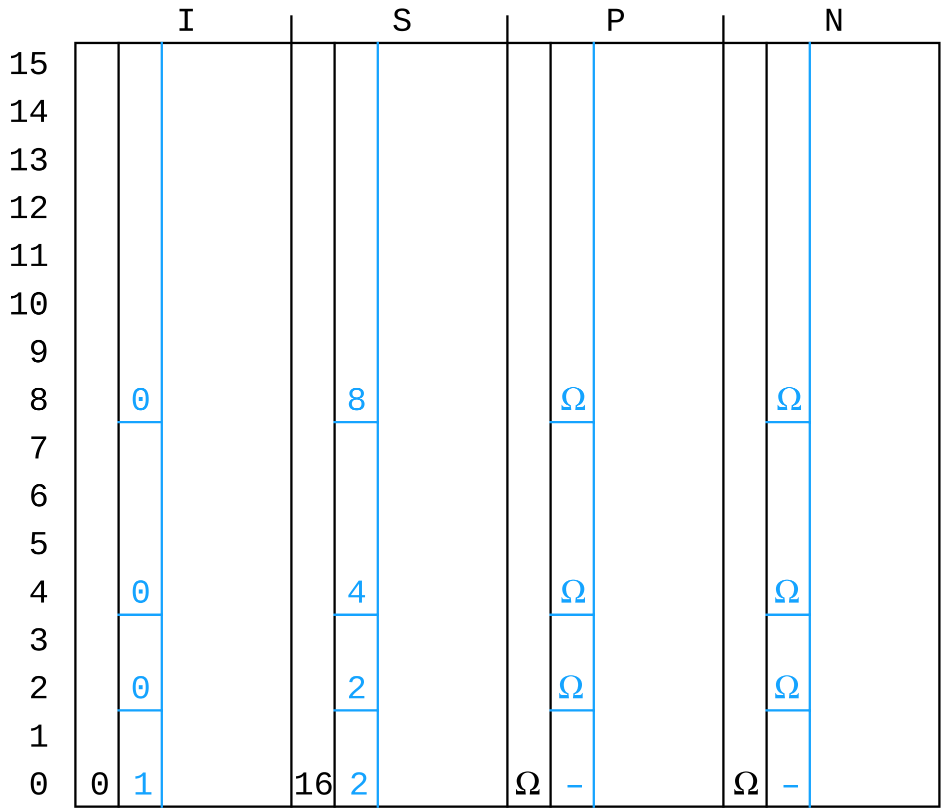
	I		S		P		N	
15								
14								
13								
12								
11								
10								
9								
8								
7								
6								
5								
4								
3								
2								
1								
0	0		16		Ω		Ω	

k	FREE[k]
0	Ω
1	Ω
2	Ω
3	Ω
4	0

initially, one block of size 16 starting at location 0 is available

EXAMPLE OF BUDDY ALGORITHM

- M = 4



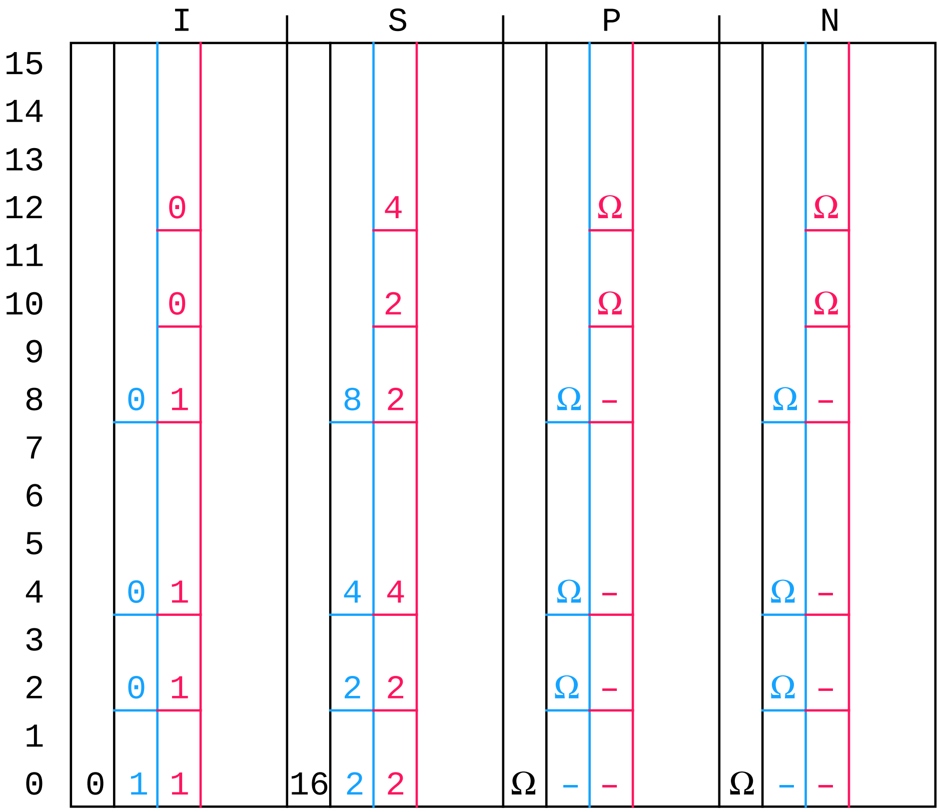
k	FREE[k]	
0	Ω	
1	Ω	2
2	Ω	4
3	Ω	8
4	0	Ω

initially, one block of size 16 starting at location 0 is available

allocate a block of size 2

EXAMPLE OF BUDDY ALGORITHM

- M = 4



k	FREE[k]		
0	Ω		
1	Ω	2	10
2	Ω	4	12
3	Ω	8	Ω
4	0	Ω	

initially, one block of size 16 starting at location 0 is available

allocate a block of size 2

allocate blocks of size 4, 2, 2 in order

EXAMPLE OF BUDDY ALGORITHM

- M = 4

	I				S				P				N			
15																
14																
13																
12			0	0			4	4			Ω	Ω			Ω	Ω
11																
10			0	0			2	2			Ω	Ω			Ω	2
9																
8		0	1	1		8	2	2		Ω	–	–		Ω	–	–
7																
6																
5																
4		0	1	1		4	4	4		Ω	–	–		Ω	–	–
3																
2		0	1	0		2	2	2		Ω	–	10		Ω	–	Ω
1																
0	0	1	1	1	16	2	2	2	Ω	–	–	–	Ω	–	–	–

k	FREE[k]		
0	Ω		
1	Ω	2	10
2	Ω	4	12
3	Ω	8	Ω
4	0	Ω	

initially, one block of size 16 starting at location 0 is available

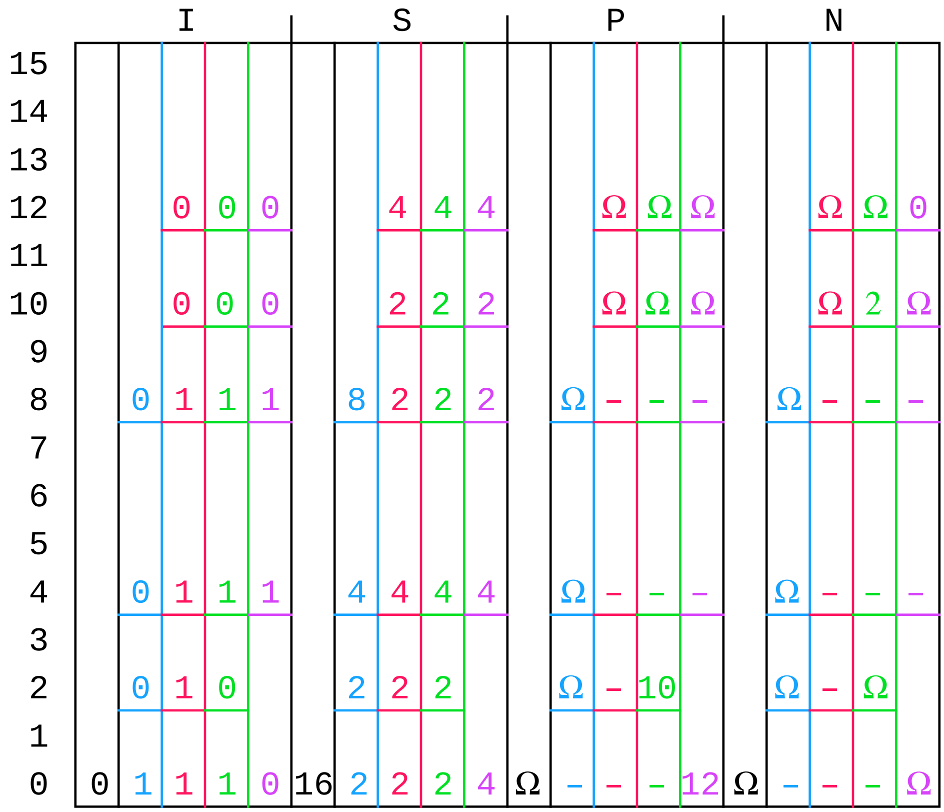
allocate a block of size 2

allocate blocks of size 4, 2, 2 in order

free the block at location 2

EXAMPLE OF BUDDY ALGORITHM

- M = 4



k	FREE[k]		
0	Ω		
1	Ω	2	10
2	Ω	4	12
3	Ω	8	Ω
4	0	Ω	

initially, one block of size 16 starting at location 0 is available

allocate a block of size 2

allocate blocks of size 4, 2, 2 in order

free the block at location 2

free the block at location 0
- merge block at 0 with its buddy at 2
- no further merging is possible as the buddy at 4 is in use

BUDDY ALGORITHM NOTES

- Assume storage runs from locations 0 to m–1
- To reserve a block of size 2^k:
  1. find smallest j for which FREE[j]≠Ω
     (assume this block starts at location n)
  2. remove the block at location n from FREE[j]
  3. ```
     while j>k do
       begin
         j←j-1;
         add block at location n+2^j to FREE[j];
       end;
     ```
- To liberate a block of size 2^k starting at location n:

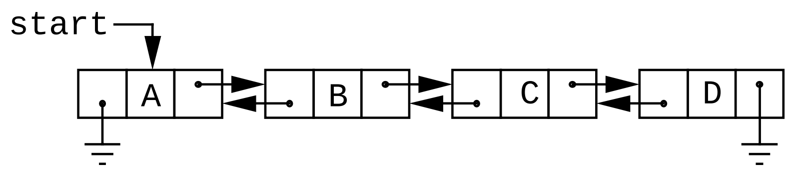
```
while k≠m and NOT(INUSE(BUDDY_k(n))) do
  begin
    remove BUDDY_k(n) from FREE[k];
    k←k+1;
    if BUDDY_k(n)<n then n←BUDDY_k(n);
  end;
```

- INUSE flag only needs to be set in first word of each reserved block
  1. all remaining elements (words) have their buddies within the same block
  2. no one outside the block will look for buddies within the block
```

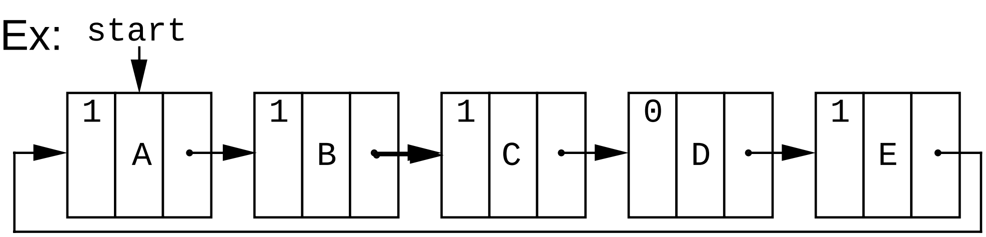
OVERFLOW

- At times, have more storage allocation requests than available memory
- Can perform garbage collection with compaction but will soon run out of memory again
- Alternatively, remove blocks to secondary storage:

1. keep a doubly-linked list of blocks in use, sorted according to frequency of use
   - whenever a block is accessed, move it to front of list
   - like a self-organizing file
   - Ex:



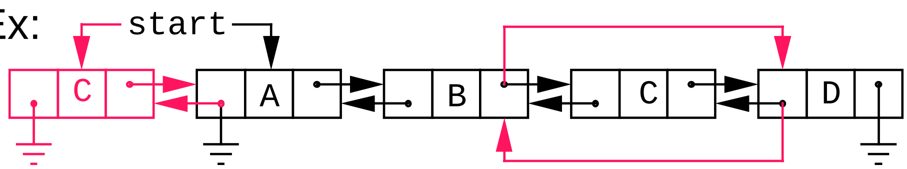
2. circular list of blocks and a recently-used bit indicating if the block was accessed since the last time blocks were removed to secondary storage
   - to remove a block, march down the list looking for a 0 and reset all 1s that were encountered to 0
   - curculating pointer ensures that a block reset to 0 will not be checked again for removal until all other blocks have been checked
   - Ex:



OVERFLOW

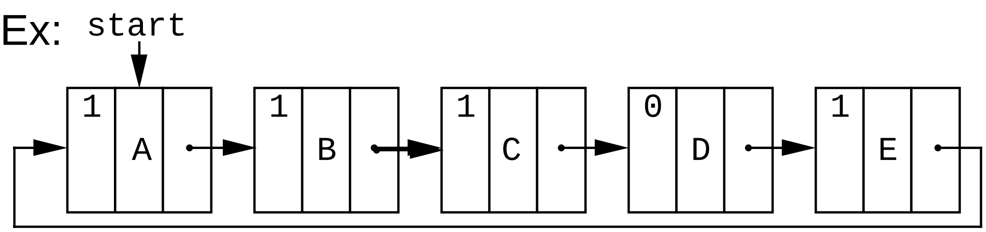
- At times, have more storage allocation requests than available memory
- Can perform garbage collection with compaction but will soon run out of memory again
- Alternatively, remove blocks to secondary storage:

1. keep a doubly-linked list of blocks in use, sorted according to frequency of use
   - whenever a block is accessed, move it to front of list
   - like a self-organizing file
   - Ex:



   - accessing C causes it to move to the front

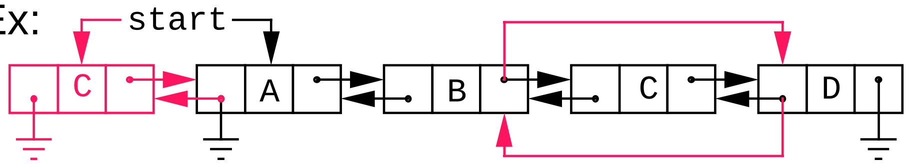
2. circular list of blocks and a recently-used bit indicating if the block was accessed since the last time blocks were removed to secondary storage
   - to remove a block, march down the list looking for a 0 and reset all 1s that were encountered to 0
   - curculating pointer ensures that a block reset to 0 will not be checked again for removal until all other blocks have been checked
   - Ex:



OVERFLOW

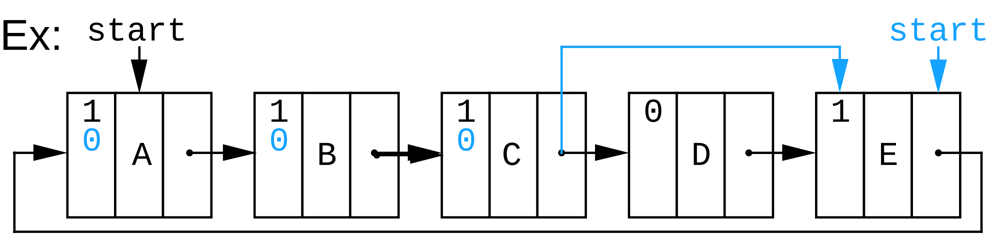
- At times, have more storage allocation requests than available memory
- Can perform garbage collection with compaction but will soon run out of memory again
- Alternatively, remove blocks to secondary storage:

1. keep a doubly-linked list of blocks in use, sorted according to frequency of use
   - whenever a block is accessed, move it to front of list
   - like a self-organizing file
   - Ex:



   - accessing C causes it to move to the front

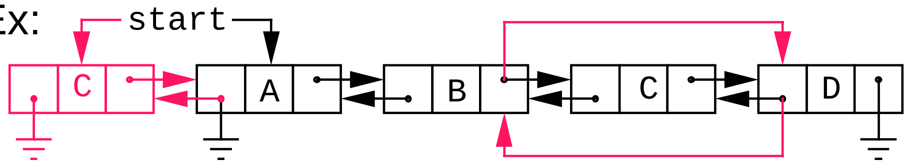
2. circular list of blocks and a recently-used bit indicating if the block was accessed since the last time blocks were removed to secondary storage
   - to remove a block, march down the list looking for a 0 and reset all 1s that were encountered to 0
   - curculating pointer ensures that a block reset to 0 will not be checked again for removal until all other blocks have been checked
   - Ex:



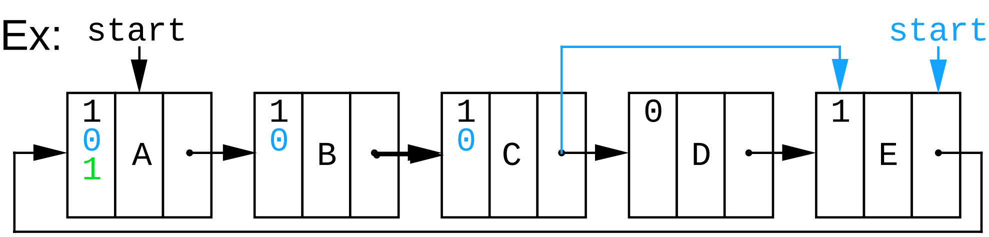
- block D is the first to be removed

OVERFLOW

- At times, have more storage allocation requests than available memory
- Can perform garbage collection with compaction but will soon run out of memory again
- Alternatively, remove blocks to secondary storage:
  1. keep a doubly-linked list of blocks in use, sorted according to frequency of use
     - whenever a block is accessed, move it to front of list
     - like a self-organizing file
     - Ex:



     - accessing C causes it to move to the front
  2. circular list of blocks and a recently-used bit indicating if the block was accessed since the last time blocks were removed to secondary storage
     - to remove a block, march down the list looking for a 0 and reset all 1s that were encountered to 0
     - curculating pointer ensures that a block reset to 0 will not be checked again for removal until all other blocks have been checked
     - Ex:

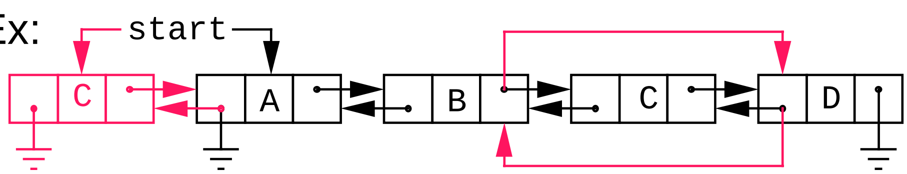


- block D is the first to be removed
- access block A

OVERFLOW

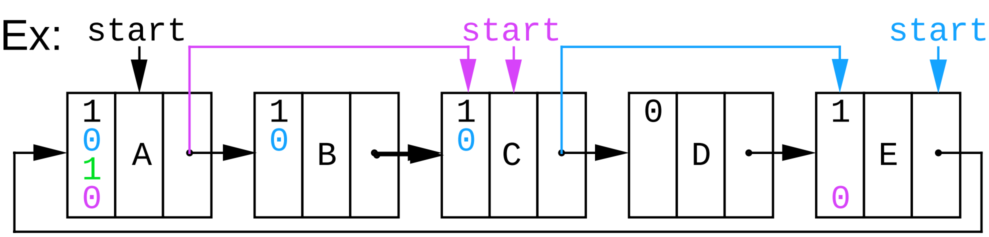
- At times, have more storage allocation requests than available memory
- Can perform garbage collection with compaction but will soon run out of memory again
- Alternatively, remove blocks to secondary storage:

1. keep a doubly-linked list of blocks in use, sorted according to frequency of use
   - whenever a block is accessed, move it to front of list
   - like a self-organizing file
   - Ex:



   - accessing C causes it to move to the front

2. circular list of blocks and a recently-used bit indicating if the block was accessed since the last time blocks were removed to secondary storage
   - to remove a block, march down the list looking for a 0 and reset all 1s that were encountered to 0
   - curculating pointer ensures that a block reset to 0 will not be checked again for removal until all other blocks have been checked
   - Ex:



- block D is the first to be removed
- access block A
- block B is removed next