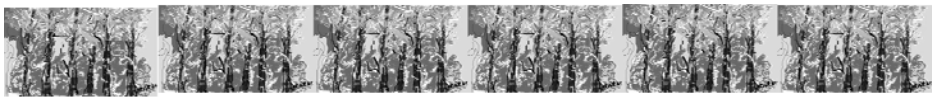


Chapter 6 – Dynamic Memory Management

Introduction to Programming Languages First Edition, 2013

Author: Arvind Bansal

ISBN: 978-146-6565142

Topics Covered

- Heap organization
- Allocation and deallocation of dynamic data objects
- Fragmentation
- Garbage collection - recycling heap memory
- Start-and-stop garbage collection
  - Mark-and-scan Algorithm
  - Copying Garbage Collection
  - Generational Garbage Collection
- Incremental garbage collection
- Reference-count garbage collection
- Concurrent garbage collection
- Real-time Garbage Collection
- Issues in Garbage Collection
- Summary

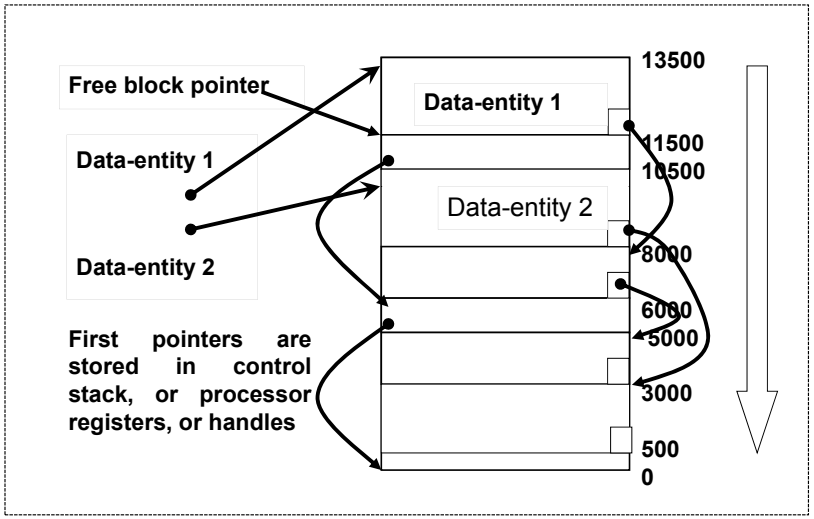
Introduction

- Memory reuse recycles released dynamic memory.
- Dynamic memory management is about allocation / release in heap.
- Dynamic memory in control stack
  - Has lifetime of the called procedure,
  - is generally allocated contiguously at the time of the procedure call
  - Is recovered by shifting TOS pointer at the end of the called procedure
- Dynamic memory in heap
  - is on demand at different times.
  - Can have lifetime beyond the called procedure.
  - Is recovered using a special class of programs called garbage collection.
  - Is needed for recursive data structures, dynamic data objects.
- Memory recycling
  - Can be done continuously, incrementally, or periodically after the heap runs out of space for allocation.

Heap Organization

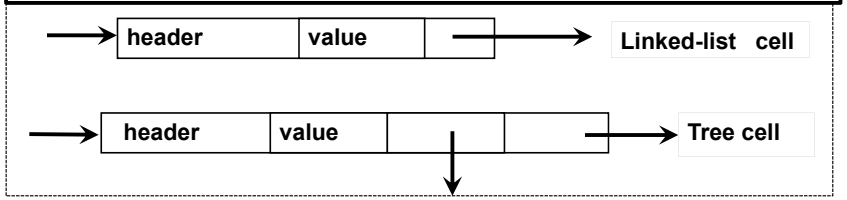
- Organized as structured linear array to allocate data objects
- A heap block has three types of information
  - Header field containing size, information field, pointer to another block
- Three types of memory blocks: active, released, and free
- Modeled as a sequence of blocks of the form
  - (*allocated, <block-size>, <start-address> <end-address>*), or
  - (*released, block-size, <start-address> <end-address>*), or
  - (*free, block-size, <start-address> <end-address>*)
- Right size block is allocated on demand
  - Free blocks can be allocated contiguously as in stack based allocation
  - Free blocks can be indexed and chained according to size of the blocks
  - Free blocks can be chained serially for first allocation
- A data object is a chain of allocated blocks
  - The first pointer to the data object is allocated in a processor register or control stack

Chained Free Blocks

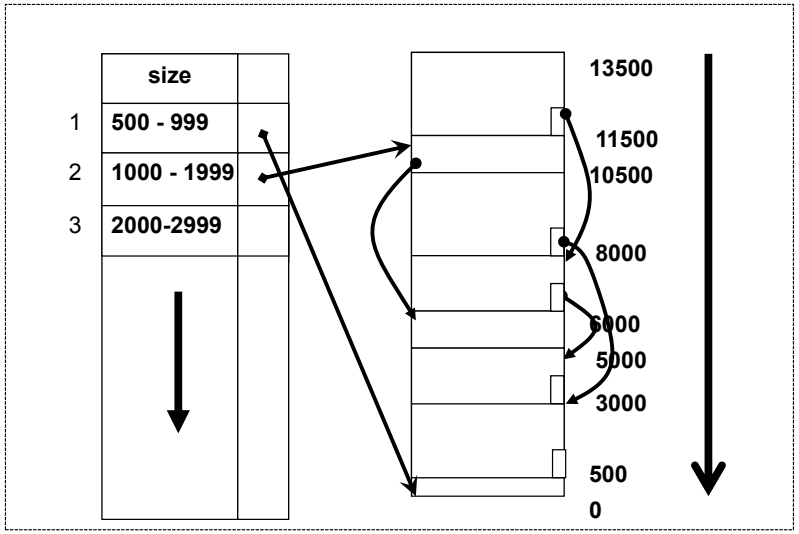


Dynamic Data Objects

- Dynamic data objects are a logical sequence of cells
- Cells have three fields: headers, value, pointer to another cell
  - Header contains the information about flags, size, number of fields in the data structure, or memory offsets from the start of the field
  - Flags can be type, released / active
- Cells can be fixed size or variable sized
- Cells are traversed using chain of pointers.
- There is an memory overhead of headers in heap based allocation for marking the fields as value / pointer / header

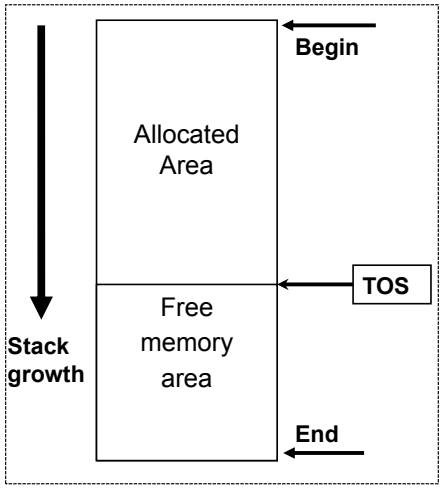


Size Based Organization of Heap



Stack Based Organization of Heap

- Heap is treated as a stack with begin and end marker and TOS
- Space allocation is done using TOS
- Memory recycling is done after TOS catches up with end marker
- Before memory allocation, the availability of requested size is verified
- Used in copying garbage collection for compaction of active data objects' space



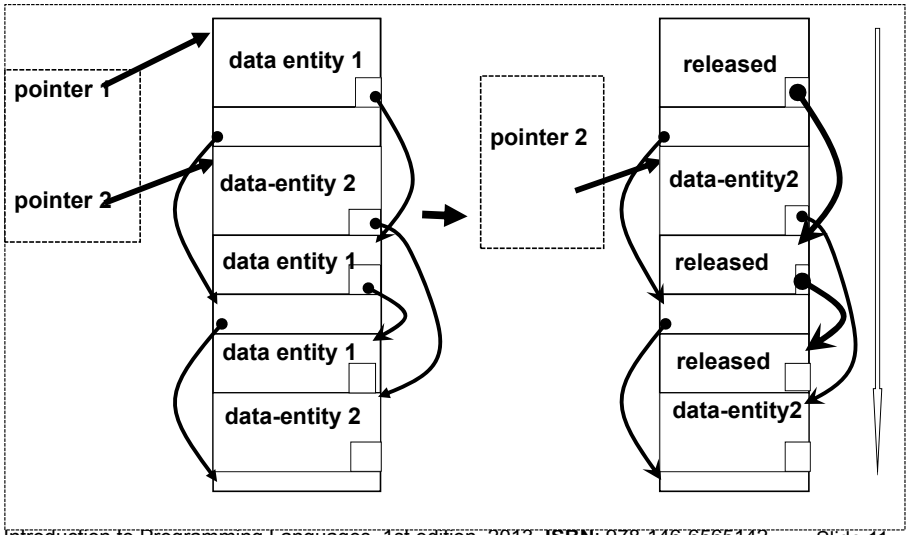
Allocation of Objects in Heap

- Data objects allocation
  - Automatically as in C#, C++, or Java or
  - On demand by programmer as in C
- First-fit allocation
  - Traverses the chain of free block, and allocates the first bigger block
  - **Disadvantage:** lopsided fragmentation in the beginning
- Next fit allocation
  - Traverses the chain of free blocks from the previous allocation point and allocates the first bigger block
  - **Advantage**: evenly distributed fragmentation
- Best fit allocation
  - The smallest free block bigger than the requested size
  - Suitable for indexed heap organization with grouped similar blocks

Deallocation of Dynamic Objects

- Deallocation depends upon a language and the lifetime of the object
  - Objects with lifetime as called procedures are deallocated automatically at the end of the called procedure.
  - Objects with lifetime beyond procedure in which they are created are deallocated by programmer's action.
- Deallocation is done by breaking the first link from the processor register or control stack to heap
  - No need to traverse every block of the deallocated data structure.
  - The deallocated may be marked released based upon garbage collection scheme or may be instantly collected.
- Inactive memory locations are candidates for memory recycling in the next cycle
  - Active cells are traversed and marked as active.
  - Untraversed cells are treated as garbage and collected.

Deallocating Objects in Heap



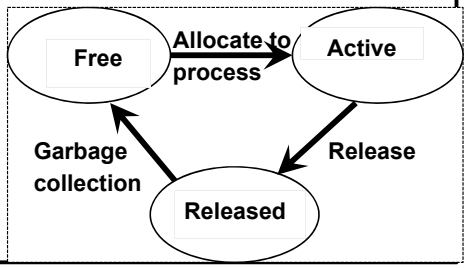
Fragmentation

- Formation of smaller block sizes that can not be individually allocated to bigger size block requests
  - With allocation and deallocation, fragments keeps increasing
- Problems with fragmentation
  - Despite available memory, fragmented blocks can not be allocated
  - Too many small fragments causes excessive overhead of accessing pages and populating cache memory
- Types of fragmentation: external or internal
  - External fragmentation is caused by left over memory block
  - Internal fragmentation is caused by fixed page based allocation
- Handling external fragmentation
  - Merge neighboring fragmented blocks
  - Compact all the blocks of the same data structure by copying
- Internal Fragmentation

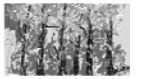
  Unavoidable in page based scheme

Garbage Collection

- Two processes: *mutator (task)* and *collector (garbage collector)*
- Three stages of memory: *free, active* and *released*
- GC is collecting released memory blocks for reallocation
- GC overhead: memory and computational is upto 20 % – 30%
- Types of garbage collection
  - Stop-and-start, incremental, continuous, concurrent, and real-time
- Stop-and-start
  - Suspends processing when GC starts
- Incremental / continuous
  - Interleaves processing and GC
- Real-time
  - Suspend GC for high priority process



Garbage Collection Approaches



- Start and stop
  - Processing stops when garbage collection starts
  - Overhead of garbage collection is 20% - 30%
  - Not suitable for real time processing: misses real-time events
- Incremental / continuous garbage collection
  - Process is interleaved with garbage collection
- Concurrent
  - Collector and mutators are executed in different threads
  - Possible if mutator and collector work in separate memory space
  - Mutator and collectors are synchronized when working on the same space
- Hard real-time
  - Gives high priority to real-time events
  - Temporarily suspend garbage collection when real-time event occurs

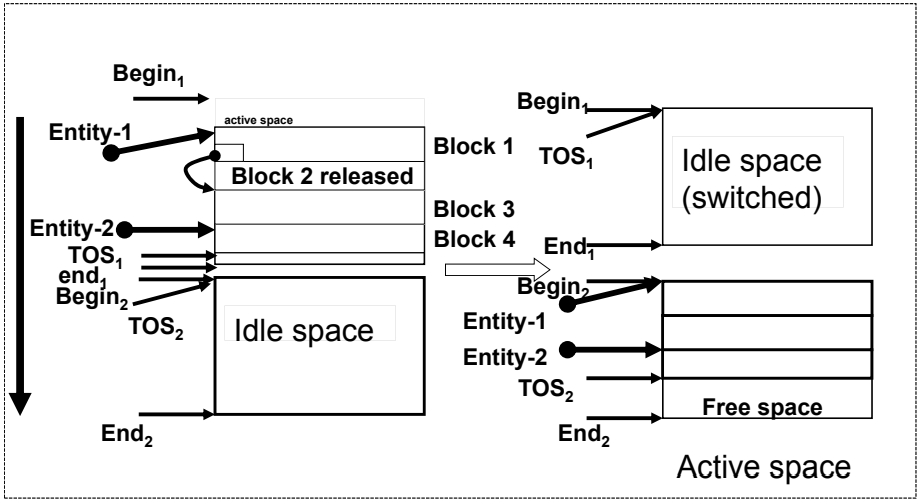
Mark-and-scan Algorithm

- Approach
  - Two phases: Mark phase and scan phase
  - Mark phase: marks active cells starting from the first pointer stored in the registers or the control stack and traversing the chain of pointers
  - Scan phase: scan all the memory and collect inactive cells
- Mark phase technique and requirement
  - Uses recursive descent from the left to right to traverse data structures
  - Set the mark bit to 1 for the active cells
  - Needs additional stack for recursive traversal during garbage collection
- Scan phase
  - Traverses sequentially cell by cell
  - Chains all the inactive cells together
  - Does not collect marked bit but resets mark bit to 0 for future cycle
- **Problems:** fragmentation; traverses active cells twice; needs additional memory for stack; stop-and-start;

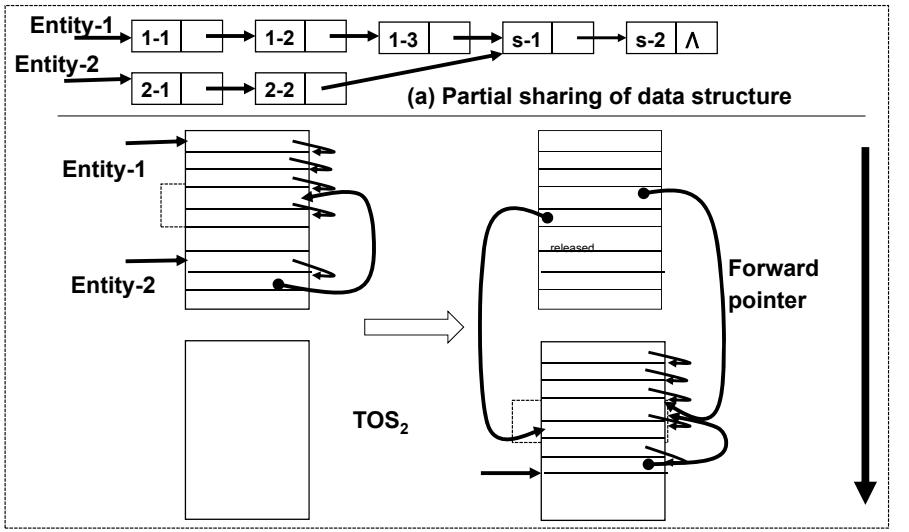
Copying Garbage Collection

- Removes fragmentation by copying one data structure at a time
- Uses stack based heap organization
- Semi-spaces: active space and idle space
  - Only active space is used for allocation
- Garbage collection process
  - GC starts when TOS(active-space) + requested-size ≥ end-marker(active-space)
  - Garbage collection copies from active space to idle space
  - One data structure at a time until all pointers to heap are consumed
  - Naïve copying garbage collection uses recursive descent
  - Uses a forward pointer for shared data structures to avoid multiple copies
- Advantage: Fragmentation is removed
- Disadvantage of Naïve Scheme
  - Stack overhead of recursive descent; 50% memory utilization
  - Start-and-stop algorithm misses real-time events

Copying Garbage Collection



Forwarding Pointer

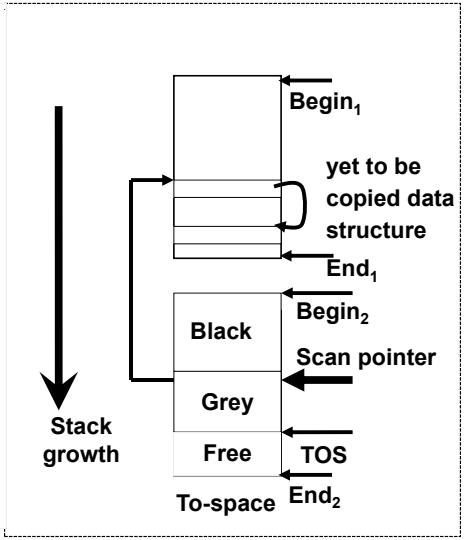


Cheney's Modification

- Goals and Advantage
  - Reduce overhead of recursive descent by removing stack overhead
- To space cells: black, grey, and white (free)
  - Black cells have been already traversed by the scan pointer
  - Grey are yet to be traversed by the scan pointer
- Technique
  - Use breadth-first search instead of depth first search
  - Use To-space as queue to avoid memory overhead of the queue
  - Use an additional scan pointer that points to the remaining data structure in from space. Scan pointer chases TOS.
  - The data is copied if the cell pointed by scan pointer is in From-space,
  - Allocation in From-space stops once garbage collection starts
  - Garbage collection is done one data structure at a time
  - GC terminates when Scan pointer ≥ TOS(TO-space) and all external pointers to heap are consumed

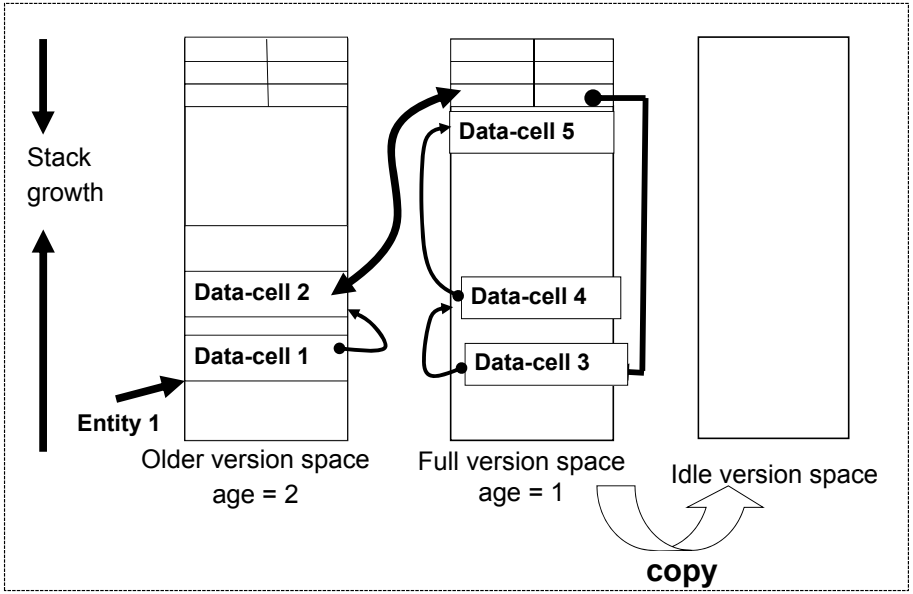
Cheney's Modifications

- From Space
  - Light green – From-space
  - Yellow – yet to be copied blocks in From-space
- To-space
  - Black – traversed by scan pointer called black space
  - Grey – yet to be traversed memory location by scan pointer
  - White – free memory locations in To-space



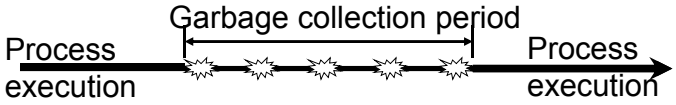
Generational Garbage Collection

- Goal: to avoid wastage of idle memory space
- Technique
  - More than two version spaces. Version spaces have an age count.
  - Only one version space is idle at a time.
  - Filled version-space is copied into an idle version space.
- Characteristics
  - A data structure may be spread over multiple version spaces.
  - Older version space do not fill up fast due to principle of locality.
  - Requires keeping tracking of pointers of split-data structures.
- Table of pointers for split data structures
  - Points to the first cell of a split data structure**.** Also keeps a pointer to the last previous cell of another version space.
  - Only the entries in the table needs to be modified during copying.
- **Disadvantage: copying and memory overhead of the table**



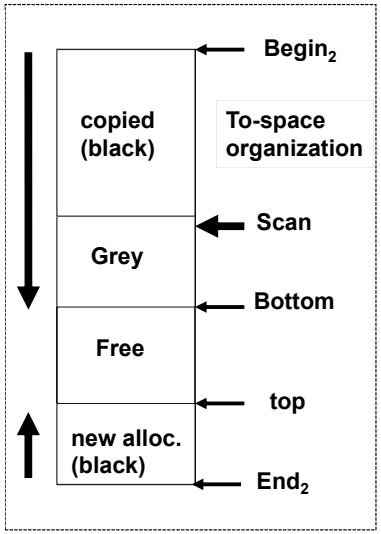
Incremental Garbage Collection

- Extension of Cheney's algorithm proposed by Baker
- Goal: continuous garbage collection instead of start-and-stop
- Technique
  - Allocate cells in To-space after GC starts
  - Collect K cells (K >> 1) for allocating every allocated cell in To-space
- Semi-space size calculation
  - Let N be the active memory locations at the time of garbage collection
  - N/K is the newly allocated cells during garbage collection
  - Free cells in To-space = N + N /K
  - Total cells in a semi-space = allocated cells + free cells = 2N + N/K
- Large K makes stop-and-start; small K GC slows down



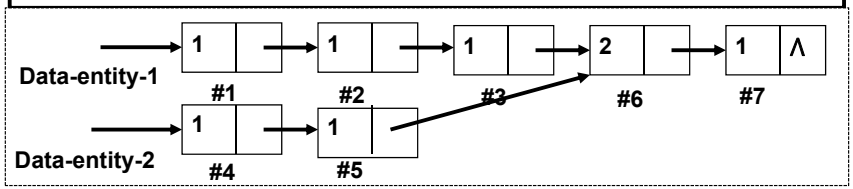
To-space during Garbage Collection

- Areas
  - **Black**: Traversed and copied
  - **Black**: newly allocated
  - **Grey**: not traversed by the scan pointer
  - **White**: free area
- Pointers
  - **Scan pointer**: points to first location of the grey area
  - **Bottom pointer**: points to the last location of the grey area
  - **Top pointer**: points to the first free location in free area
- Termination
  - **Scan pointer =< Bottom pointer and all pointers to the heap are consumed**

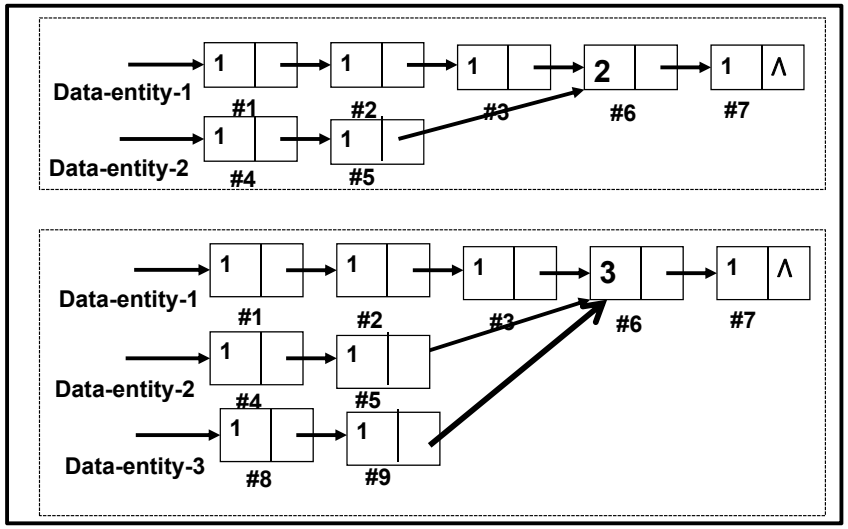


Reference-count Garbage Collection

- Continuous garbage collection useful for data structures with shared memory blocks
- Advantage: avoids dangling reference problem and recycling of memory while in use by other data structure
- Technique
  - Keep a reference count with every cell
  - Increment the ref-count when additional pointer is added
  - Decrement the ref-count of all the constituting cells after deletion of a data structure until the first shared cell
  - Garbage collect when reference count = 0
  - Do not garbage collect when reference count ≥ 1

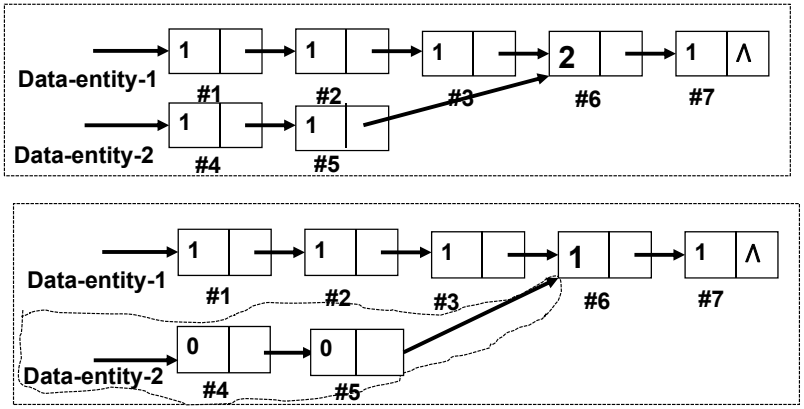


Insertion in Reference Count



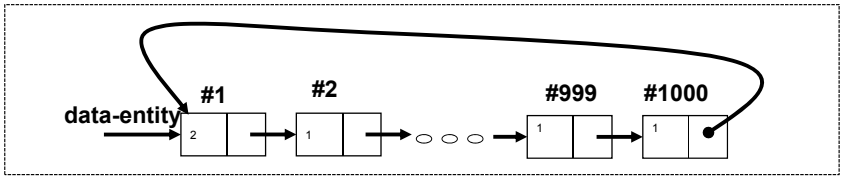
Deletion in Reference Count

- Ref-count of all cells in the deleted data structure are decremented until first cell with ref-count > 1 or null ptr.
- Cells with ref-count == 0 are recycled



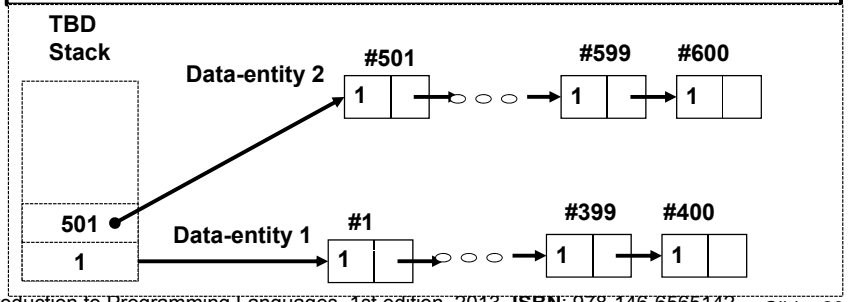
Handling Cyclic Data-structures

- Handling cyclic data structures
  - Reference-count of cyclic data-structure's start cell ≥ 2
  - Deleting will decrement the reference count by only one
  - Detecting cycles is computationally expensive for large cycles
- Practical solution
  - Only 2% of memory is lost by ignoring the cycles.
  - It is better to ignore the cycles and not to recollect them



To be Decremented Stack

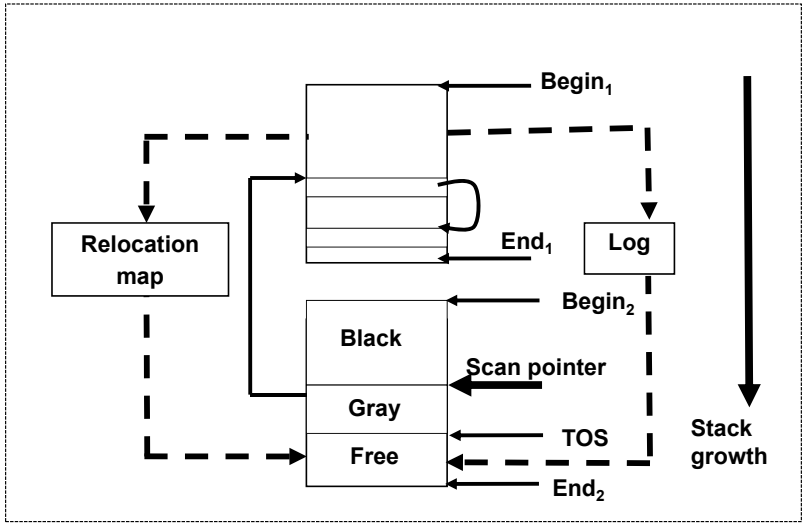
- Handling overhead of deleting large data structures
  - Use a to-be-decremented stack that stores the address of the first cell of the yet-to-recycled deleted data structures
  - The cells' address is popped on demand, and the reference-count is decremented by one
  - Those cells with reference-count == 0 are collected until the shared cell with reference-count ≥ 1



Concurrent Copying GC

- Allows finer grain than incremental garbage collection
  - Runs collector and mutator in two different threads.
  - Independent action in memory not shared by the collectors and mutator.
  - Synchronize the actions on memory shared by collectors and mutators.
- Concurrent copying garbage collection
  - Allows concurrency as well as compaction of the data structures.
  - Uses special data structures: **mutation log** and **Relocation map**
  - Relocation map stores the copied entries, and is of the form (from-location, to-location). Implemented using a hash-table.
  - Mutation log keeps all the changes in the From-space during garbage collection to be incorporated in To-space by collector.
  - No need of forward pointer due to the presence of relocation map.

Concurrent Copying GC



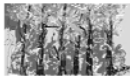
Issues in Garbage Collection

- Overhead of garbage collection is 20 – 30%
- Stop-and-start GC misses real-time events
- Memory leak due to
  - Not collecting cyclic data structures due to excessive overhead
  - Mix up of data and pointer due to lack of tag separating data and pointer in garbage collectors for traditional languages as C
- Mix-up of data and pointer during garbage collection
  - Pointer treated as data causes lack of marking of active cells and improper garbage collection causing memory corruption
  - Data cell treated as pointer causes memory leak
- Techniques used to separate data and pointers
  - Word-alignment of data; and
  - Initializing data to some padded value

Summary I

- Heap is a common memory area visible to all subprograms for the allocation of recursive and dynamic data structures
- Free block organization in heap can be single chain based, index based, or stack based
- Allocation scheme can be first fit, best fit, or next fit
- Logical data structures are a chained blocks in heap arranged in chronological order of allocation
- First pointer to heap is stored in processor register or control stack
- Deallocation is done by removing the first pointer to heap
- Memory in heap has three states: active, released and free
- Garbage collection can be start and stop or real-time
- Real-time can be achieved by
  - Incremental garbage collection, continuous garbage collection, concurrent garbage collection, or hard real time garbage collection

Summary II



- Data structure traversal can be done using depth-first (using stack) or breadth-first (using queue)
  - Depth first traversal has additional overhead of stack. Breadth first search uses To-space to alleviate the overhead of the queue
- Mark-and-scan algorithm
  - Mark phase marks active cells, and scan phase collects remaining cells
  - Has limitations of: 1) stack overhead; 2) traversing active cells twice; 3) stop-and-start; and 4) fragmentation
- Copying garbage collection removes fragmentation by copying one logical data structure in physically contiguous locations
- Cheney's improvement removes stack overhead by performing breadth-first search and use of To-space as queue
- Baker's algorithm provides incremental garbage collection by copying K (K >> 1) cells for every newly allocated cell during GC

Summary III

- Generational garbage collection uses multiple version spaces.
  - overhead of keeping connectivity tables to connect split data structures
  - Older data structures are not recycled
- Reference-count garbage collection is
  - Continuous garbage collection and is useful for shared data-space
  - Has problem with cyclic data structures
  - Memory overhead of reference counts
  - Computational overhead of traversing deleted data structure, uses TBD-stack for demand based recycling
- Concurrent garbage collection uses multiple threads, atomicity and synchronization for shared space.
- Hard real-time garbage collection
  - Uses priority scheduling for real-time tasks in addition to concurrent garbage collection