Run-time Environments
- Part 3

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design

Outline of the Lecture – Part 3

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Heap Memory Management

- Heap is used for allocating space for objects created at run time
  - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
  - *malloc()* and *free()* in C programs
  - *new()* and *delete()* in C++ programs
  - *new()* and garbage collection in Java programs
- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)

Memory Manager

- Manages heap memory by implementing mechanisms for allocation and deallocation, both manual and automatic
- Goals
  - Space efficiency: minimize fragmentation
  - Program efficiency: take advantage of locality of objects in memory and make the program run faster
  - Low overhead: allocation and deallocation must be efficient
- Heap is maintained either as a doubly linked list or as bins of free memory chunks (more on this later)

Allocation and Deallocation

- In the beginning, the heap is one large and contiguous block of memory
- As allocation requests are satisfied, chunks are cut off from this block and given to the program
- As deallocations are made, chunks are returned to the heap and are free to be allocated again (*holes*)
- After a number of allocations and deallocations, memory is fragmented and not contiguous
- Allocation from a fragmented heap may be made either in a *first-fit* or *best-fit* manner
- After a deallocation, we try to *coalesce* contiguous holes and make a bigger hole (free chunk)

Heap Fragmentation

busy	free	busy	free	busy	busy	free
100K	50K	20K	50K	200K	30K	50K

- To begin with the whole heap is a single chunk of size 500K bytes
- After a few allocations and deallocations, there are holes
- In the above picture, it is not possible to allocate 100K or 150K even though total free memory is 150K

First-Fit and Best-Fit Allocation Strategies

- The *first-fit* strategy picks the **first** available chunk that satisfies the allocation request
- The *best-fit* strategy searches and picks the smallest (**best**) possible chunk that satisfies the allocation request
- Both of them chop off a block of the required size from the chosen chunk, and return it to the program
- The rest of the chosen chunk remains in the heap

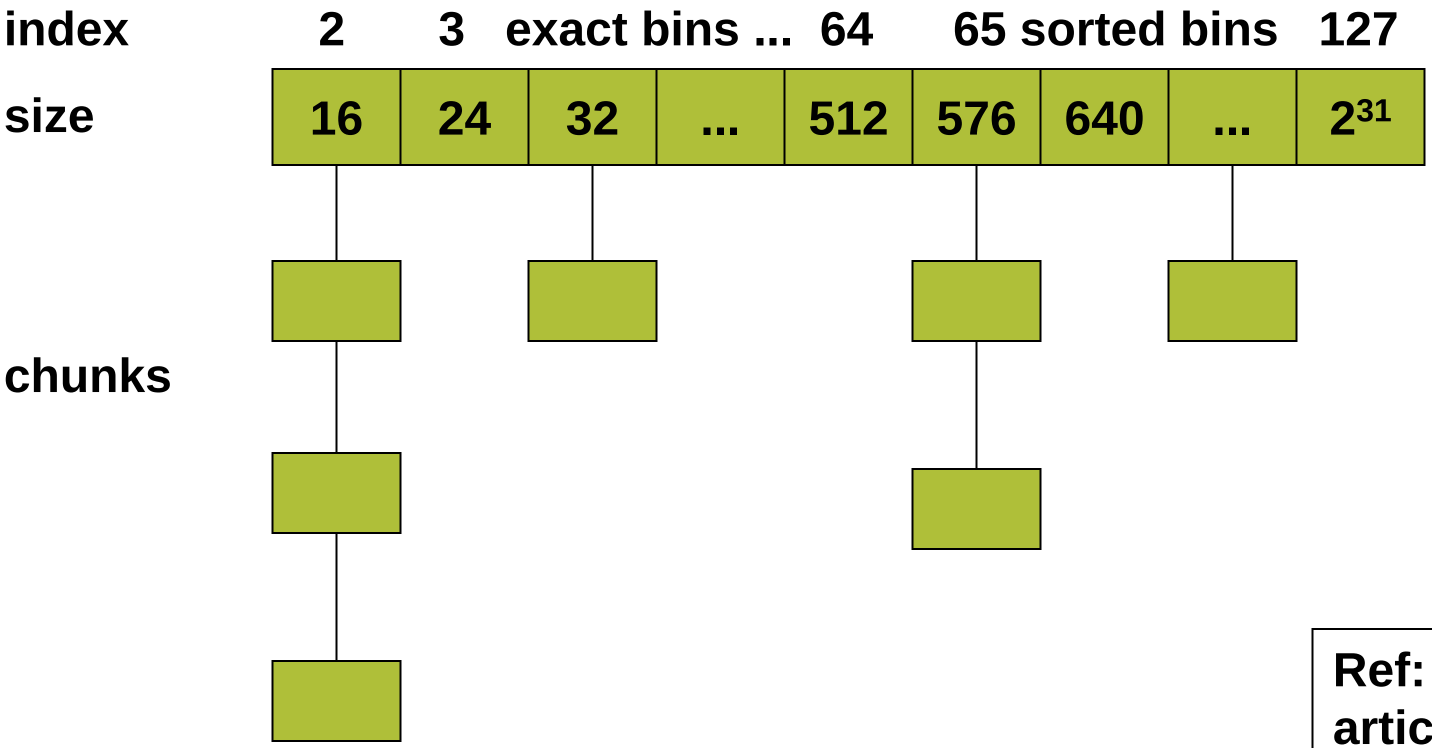
First-Fit and Best-Fit Allocation Strategies

- Best-fit strategy has been shown to reduce fragmentation in practice, better than first-fit strategy
- *Next-fit* strategy tries to allocate the object in the chunk that has last been split
  - Tends to improve speed of allocation
  - Tends to improve spatial locality since objects allocated at about the same time tend to have similar reference patterns and life times (cache behaviour may be better)

Bin-based Heap

- Free space organized into *bins* according to their sizes (**Lea Memory Manager in GCC)**
  - Many more bins for smaller sizes, because there are many more small objects
  - A bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes
  - Then approximately logarithmically (double previous size)
  - Within each "small size bin", chunks are all of the same size
  - In others, they are ordered by size
  - The last chunk in the last bin is the *wilderness chunk*, which gets us a chunk by going to the operating system

Bin-based Heap – An Example


index 2 3 exact bins ... 64 65 sorted bins 127

size 16 24 32 ... 512 576 640 ... 2^{31}

chunks

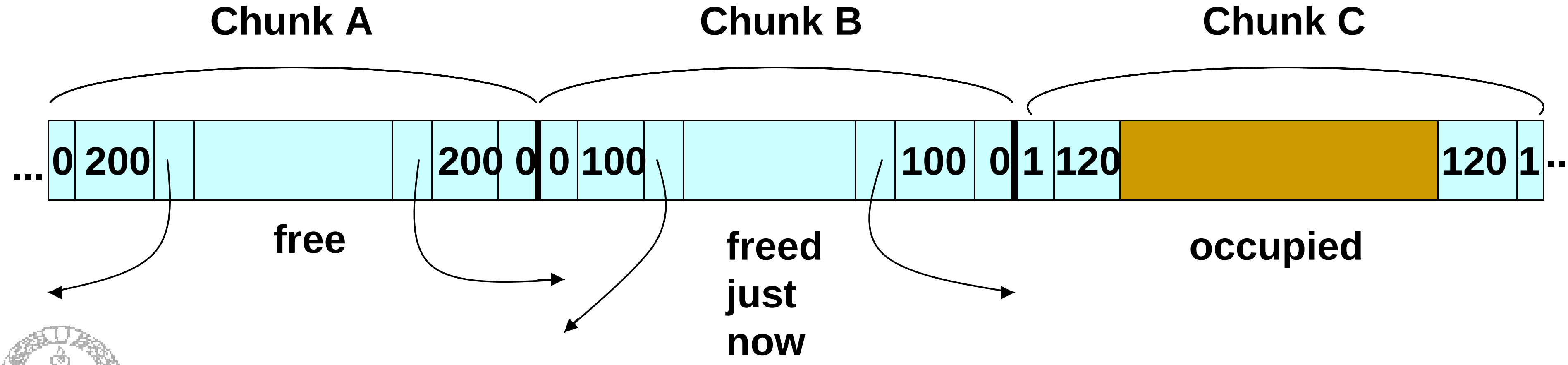
Ref: From Lea's article on memory manager in GCC

Managing and Coalescing Free Space

- Should coalesce adjacent chunks and reduce fragmentation
  - Many small chunks together cannot hold one large object
  - In the Lea memory manager, no coalescing in the exact size bins, only in the sorted bins
  - *Boundary tags* (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
  - A *doubly linked list* of free chunks

Boundary Tags and Doubly Linked List

3 adjacent chunks. Chunk B has just been deallocated and returned to the free list. Chunks A and B can be merged, and this is done just before inserting it into the linked list. The merged chunk AB may have to be placed in a different bin.



Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data
- Both are serious and hard to debug

Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
  - Then, size and pointer fields of objects can be determined by the GC
  - Languages in which types of objects can be determined at compile time or run-time are type safe
    - Java is type safe
    - C and C++ are not type safe because they permit type casting, which creates new pointers
    - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

Reachability of Objects

- Similarly, an object that becomes *unreachable* can cause more objects to become unreachable
- A garbage collector periodically finds all unreachable objects by one of the two methods
  - Catch the transitions as reachable objects become unreachable
  - Or, periodically locate all reachable objects and infer that all *other* objects are unreachable

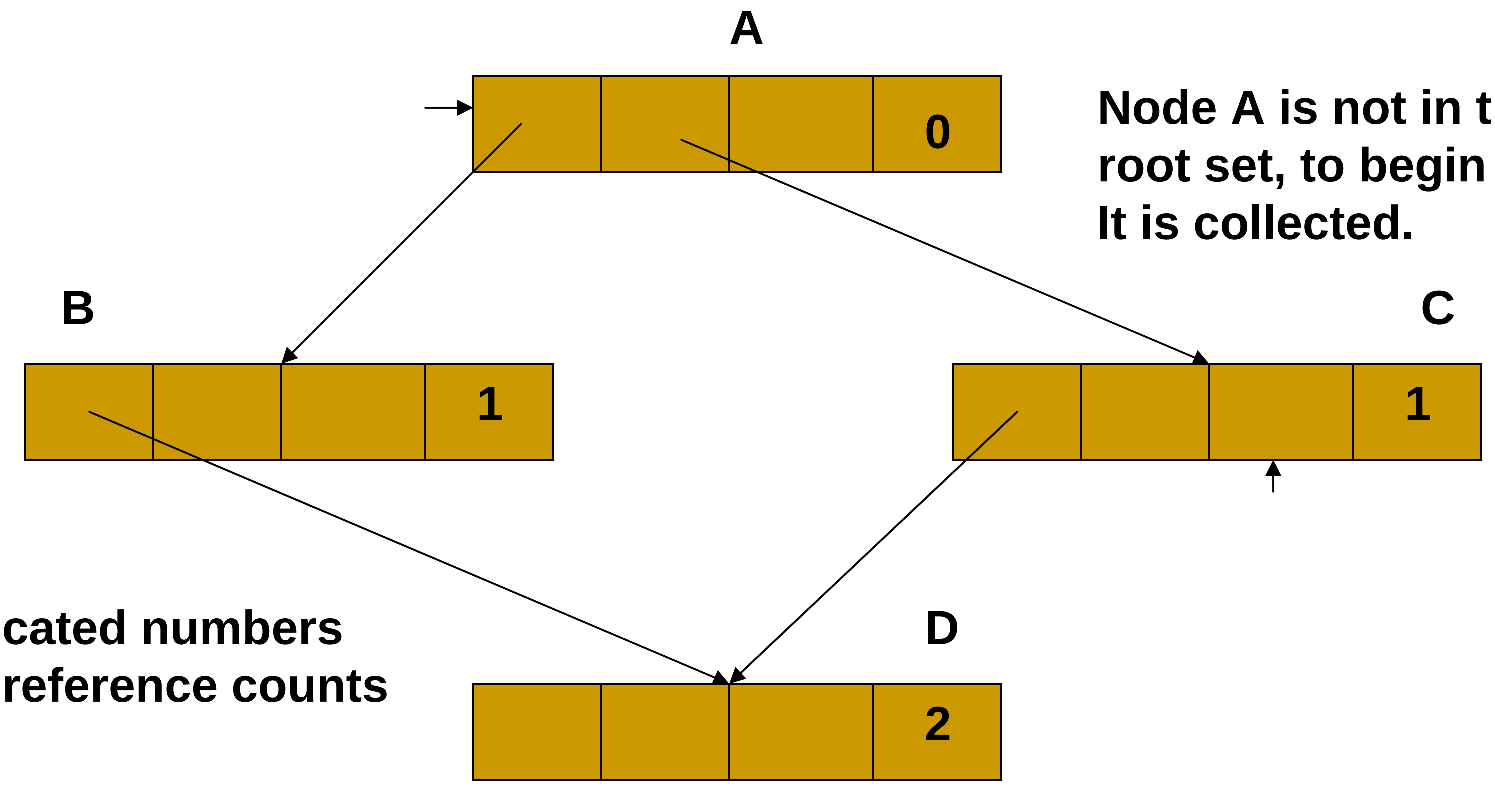
Reference Counting Garbage Collector

- This is an approximation to the first approach mentioned before
- We maintain a count of the references to an object, as the mutator (program) performs actions that may change the reachability set
- When the count becomes zero, the object becomes unreachable
- Reference count requires an extra field in the object and is maintained as below

Maintaining Reference Counts

- *New object allocation.* ref_count=1 for the new object
- *Parameter passing.* ref_count++ for each object passed into a procedure
- *Reference assignments.* For u:=v, where u and v are references, ref_count++ for the object *v, and ref_count-- for the object *u
- *Procedure returns.* ref_count-- for each object pointed to by the local variables
- *Transitive loss of reachability.* Whenever ref_count of an object becomes zero, we must also decrement the ref_count of each object pointed to by a reference within the object

Reference Count Manipulation

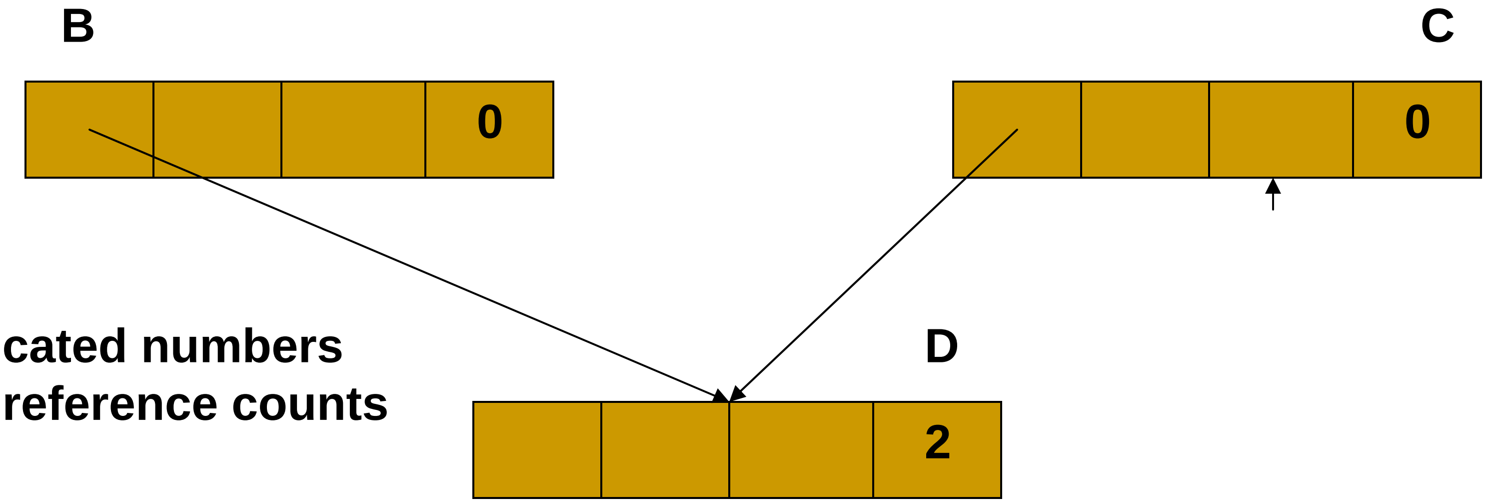


Node A is not in the root set, to begin with. It is collected.

Indicated numbers are reference counts

Reference Count Manipulation

Nodes B and C now are unreachable.
They are collected

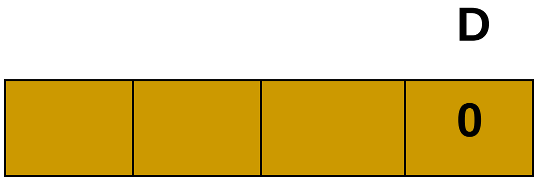


Indicated numbers
are reference counts

Reference Count Manipulation

Node D is now unreachable.
It is collected too.

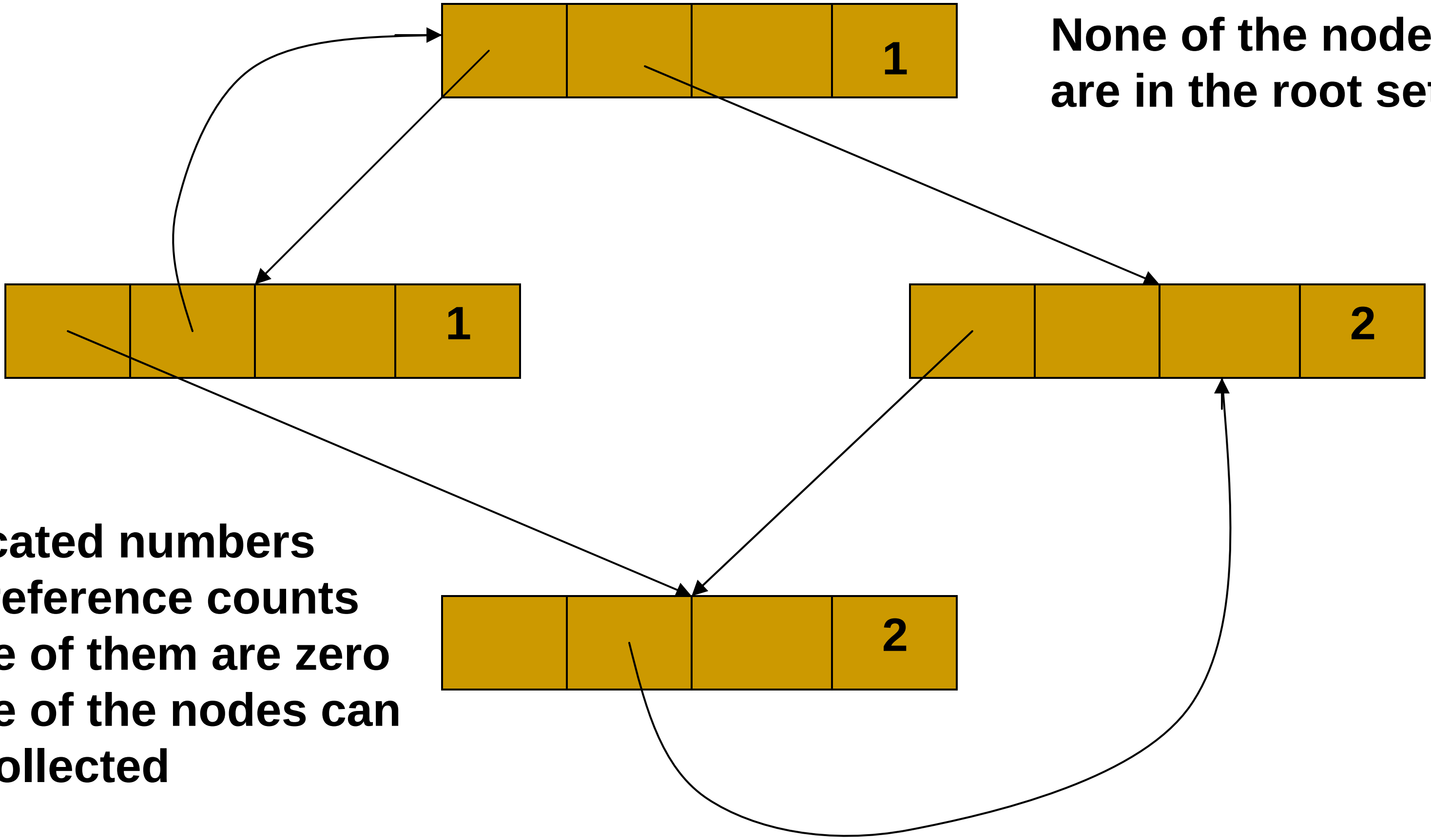
Indicated number is the reference count



Reference Counting GC: Advantages and Disadvantages

- High overhead due to reference maintenance
- Cannot collect unreachable cyclic data structures (ex: circularly linked lists), since the reference counts never become zero
- Garbage collection is incremental
  - overheads are distributed to the mutator’s operations and are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applications, where long and sudden pauses are unacceptable

Unreachable Cyclic Data Structure



None of the nodes are in the root set

Indicated numbers
are reference counts
None of them are zero
None of the nodes can
be collected