Run-time Environments - 2

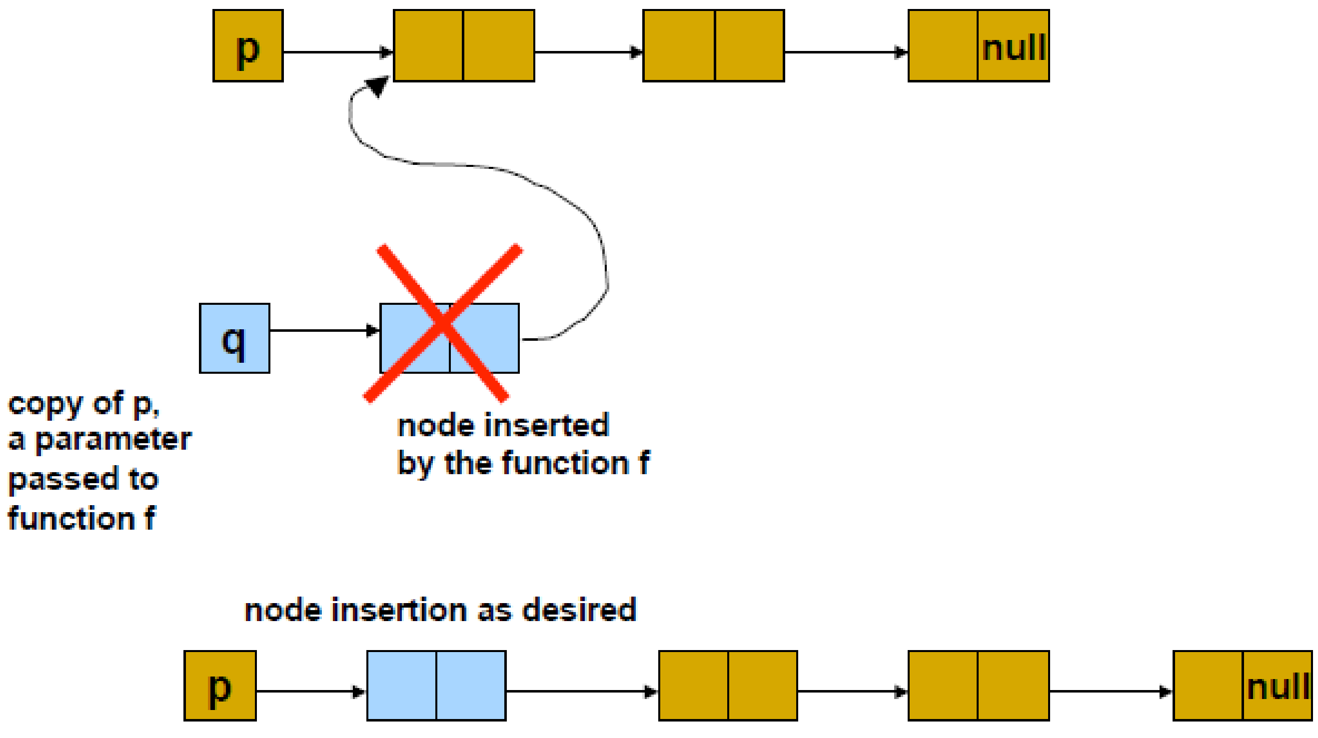
Some material is from Prof Y N Srikant (IISc Bangalore), available at NPTEL.

- Parameters can be passed to a function in various ways
  - Call by value
  - Call by reference
  - Call by value result
  - Call by name

Parameter Passing Methods
- Call-by-value

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
  - Thus, there is no way to change the actual parameters.
  - Found in C and C++
  - C has only call-by-value method available
    - Passing pointers does not constitute call-by-reference
    - Pointers are also copied to another location
    - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers

Problem with Call-by-Value



Parameter Passing Methods
- Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable
- The **address** of the variable (or the temporary) is passed to the called procedure
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure
- Found in C++ and Java

Call-by-Value-Result

- ***Call-by-value-result*** is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable
- Found in Ada

Difference between Call-by-Value, Call-by-Reference, and Call-by-Value-Result

```
int a;
void Q()
    { a = a+1; }
void R(int x);
    { x = x+10; Q(); }
main()
    { a = 1; R(a); print(a); }
```

call-by-value	call-by-reference	call-by-value-result
2	12	11

Value of a printed

Note: In Call-by-V-R, value of x is copied into a, when proc R returns. Hence a=11.

Parameter Passing Methods - Call-by-Name

- Use of a call-by-name parameter implies a **textual** substitution of the formal parameter name by the **actual** parameter
- For example, if the procedure

  void R (int X, int I);

  { I = 2; X = 5; I = 3; X = 1; }

  is called by R(B[J*2], J)

  this would result in (effectively) changing the body to

  { J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }

  just before executing it

Call by name

- Found in Algol and in some functional languages.

Example of Using the Four Parameter Passing Methods

```
void swap (int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
} /*swap*/
...
{ i = 1;
  a[i] =10; /* int a[5]; */
  print(i,a[i]);
  swap(i,a[i]);
  print(i,a[1]); }
```

- Results from the 4 parameter passing methods (print statements)

call-by-value	call-by-reference	call-by-val-result	call-by-name
1 10 1 10	1 10 10 1	1 10 10 1	1 10 error!

Reason for the error in the Call-by-name Example

The problem is in the swap routine

```
temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */
```

Overlapped Variable Storage for Blocks in C

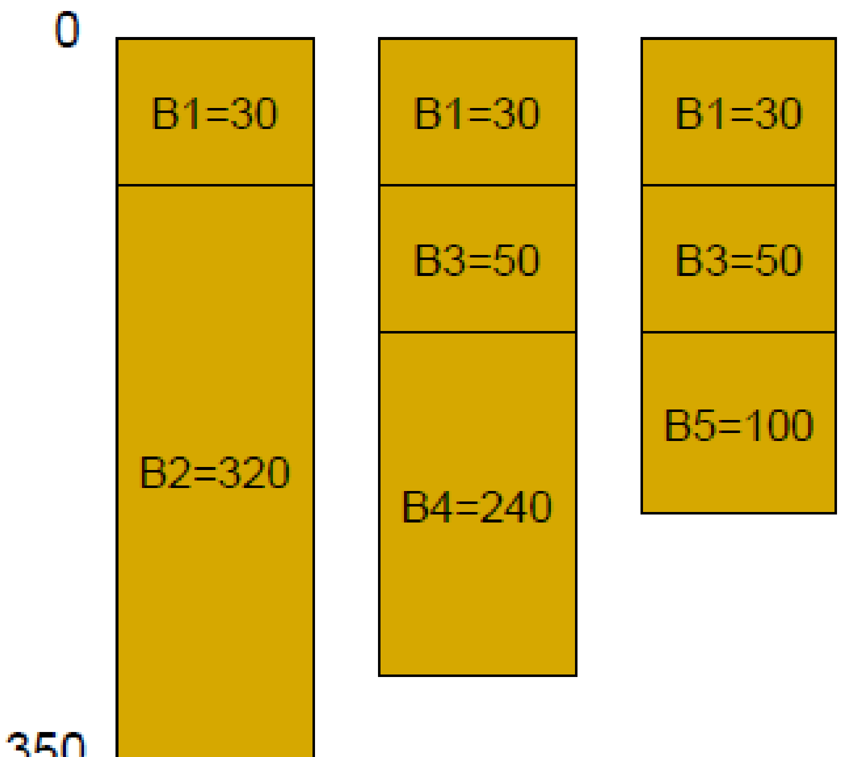
```
int example(int p1, int p2)
B1 { a,b,c; /* sizes - 10,10,10;
              offsets 0,10,20 */
    ...
    B2 { d,e,f; /* sizes - 100, 180, 40;
                  offsets 30, 130, 310 */
         ...}
    B3 { g,h,i; /* sizes - 20,20,10;
                  offsets 30, 50, 70 */
         ...
         B4 { j,k,l; /* sizes - 70, 150, 20;
                       offsets 80, 150, 300 */
              ... }
         B5 { m,n,p; /* sizes - 20, 50, 30;
                       offsets 80, 100, 150 */
              ... }
    }
}
```

Overlapped storage

Overlapped storage

Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
30+max(320, (50+240)) =
30+max(320,290) = 350

Overlapped Variable Storage for Blocks in C (Ex.)



Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
30+max(320, (50+240)) =
30+max(320,290) = 350

7.4 Heap Management

- Data on heap outlives termination of the procedure that created it, unless deleted.
- One way to share data among various procedures.
  - Eg: Working with linked lists. One procedure adds, other deletes from the same.
- We see the memory manager, the subsystem that allocates and deallocates space in the heap.

How this happens/achieved?

- When we ask for heap memory, like using
  z = (int *)malloc(4*sizeof(int));
  assuming int requires 4 bytes the memory needed is 16 bytes.
- After finding size required. A function call like
  z = memory_manager(16); is made.
- Now, memory manager knows how heap is organized. So will sanction the request (deos necessary adjustments to the data structures, etc), or will return NULL value.
- So, memory_manager( ) is part of every program.

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 management

- Memory management would be simpler if
  - All allocation requests were for chunks of same size. Eg: For Lisp this is true (a two pointer cell)
  - Storage is released predictably, say, first-allocated first-deallocated.
    - A circular queue like structure will solve our problem of memory management! And this is the optimal, in this scenario.
  - In most cases, neither of these holds.

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)

7.4.2 The Memory Hierarchy of a Computer

- Programmer, in general is unaware of the memory subsystem (types of memory, mechanisms of allocation etc).
- The time taken to execute an instruction can vary significantly, since the time taken to access different parts of memory can vary from nanoseconds to milliseconds.
- Data-intensive programs, therefore benefit from the knowledge of the memory subsystem.

Memory hierarchy

- One can take advantage of the phenomenon of “locality” – the nonrandom behavior of typical programs.
- The variation in memory access times is due to the fundamental limitation in hardware technology.
  - We can build small and fast storage, or large and slow storage. But not large and fast!

- All modern computers arrange their storage as a memory hierarchy.
  - Small & fast are typically closer to the processor than the larger&slow ones.

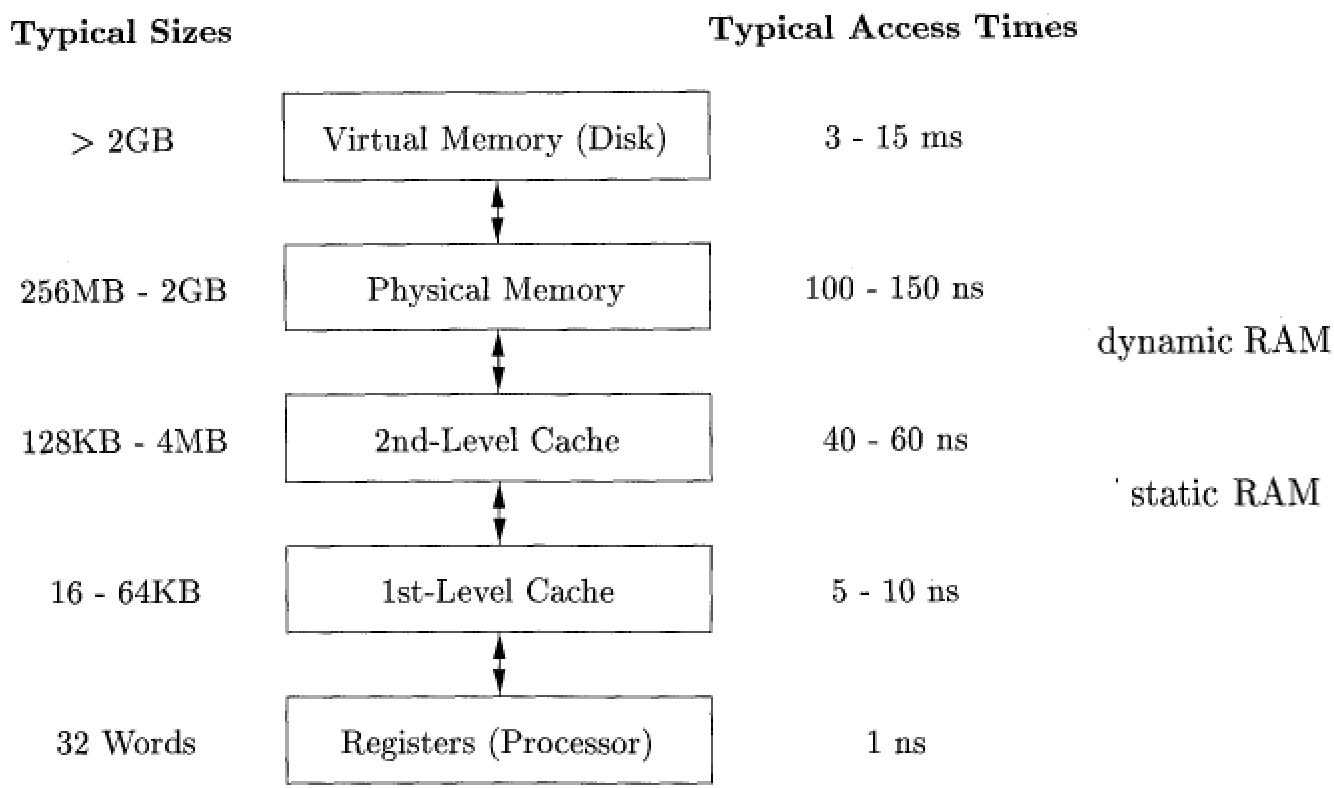


Figure 7.16: Typical Memory Hierarchy Configurations

7.4.3 Locality in Programs

- Locality means programs spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data.
- Temporal locality. Memory accessed now, in near future is again accessed.
- Spatial locality. Nearby memory cells are accessed within a short period of time.

- It is known that, generally programs spend 90% of their time executing 10% of the code.
  - Programs contain, often many instructions that are never executed. The libraries or tools are included which are never fully used.
  - Most of the code is to avoid errors/exceptions, which in general scenarios, does not occur.
  - Innermost loops are executed many many times than outer loops or other code.

Optimization using the memory hierarchy

- Memory hierarchy should be transparent to the programmer, unless in a relatively low level HLL.
- Placing frequently used data in fast memory can improve the overall speed of the program.
- Often this is difficult for the compiler (requires semantic analysis of the program which is, in general, beyond the scope of the compiler).
- Compilers often simply believes in “locality”.

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 becomes fragmented and is 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 been split recently
  - 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)

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

Lea memory manager – good for best fit strategy

- Separate space into bins, according to their sizes.

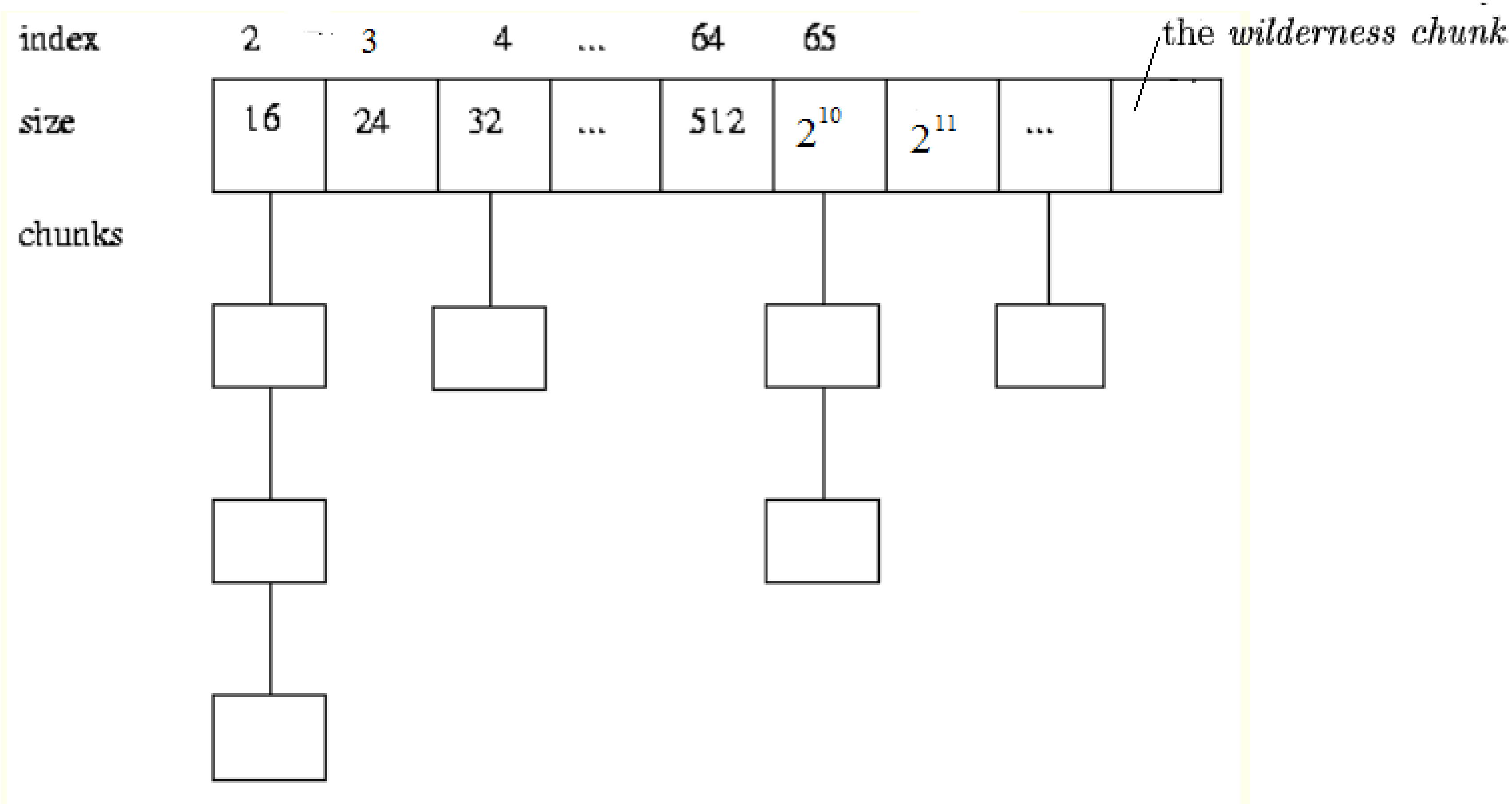
Fixed size chunk bins

- There is a bin for 16 byte chunks.
- Then, a bin for 24 byte chunks.
- So on, multiples of 8 bytes…. Upto 512 bytes bin.

Variable size chunk bins

- After this, chunk size doubles for next bins. A bin for 1024byte chunks (to less than 2k size chunks), then a bin for 2k size, then for 4k size, so on. (where this is going to end depends on how much heap is available to begin with).
- The last one is the wilderness chunk, this chunk is treated by Lea as the largest-sized bin (which is obtained from OS on request).

Lea memory manager -- illustration



Coalescing adjacent chunks is a problem with Lea memory

- Two adjacent holes can be coalesced in to bigger hole.
- With bitmaps and movement of chunks across bins this can be achieved in Lea memory manager. This is done for variable sized bins.
- There are two data structures which support coalescing of adjacent holes easily.

Boundary tags and doubly linked list

- Boundary Tags: At both ends of a chunk we keep a bit to indicate whether the chunk is free (0) or used (1). Adjacent to this bit is a count that tells the size of the chunk.
- A doubly linked, embedded free list: The free chunks are linked in a doubly linked list.
  - This allows searching for a free chunk, and best fit can be done (by searching the entire list).

Example 7.10: Figure 7.17 shows part of a heap with three adjacent chunks, A, B, and C. Chunk B, of size 100, has just been deallocated and returned to the free list. Since we know the beginning (left end) of B, we also know the end of the chunk that happens to be immediately to B's left, namely A in this example. The free/used bit at the right end of A is currently 0, so A too is free. We may therefore coalesce A and B into one chunk of 300 bytes.

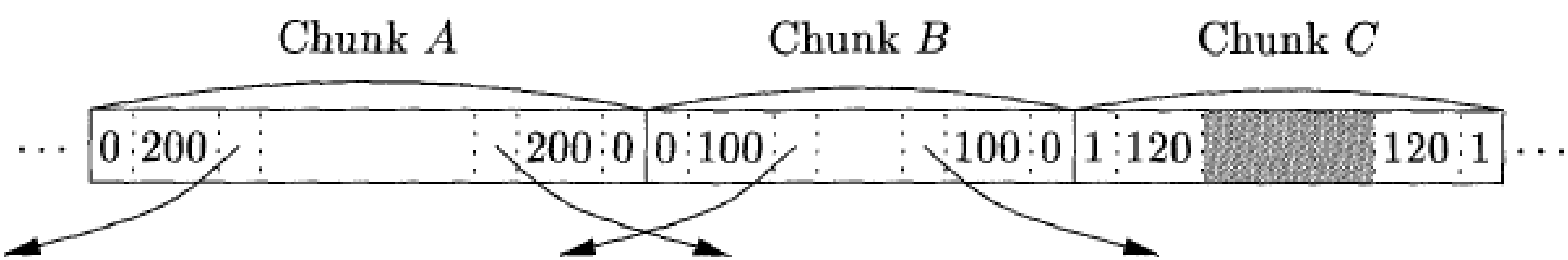


Figure 7.17: Part of a heap and a doubly linked free list

- Note, if we always coalesce chunks as soon as we can, then there can never be two adjacent free chunks.
- So when a chunk is freed, it is enough to check the two adjacent chunks of this, to do the coalescing.

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
- Solution: automatic garbage collection

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

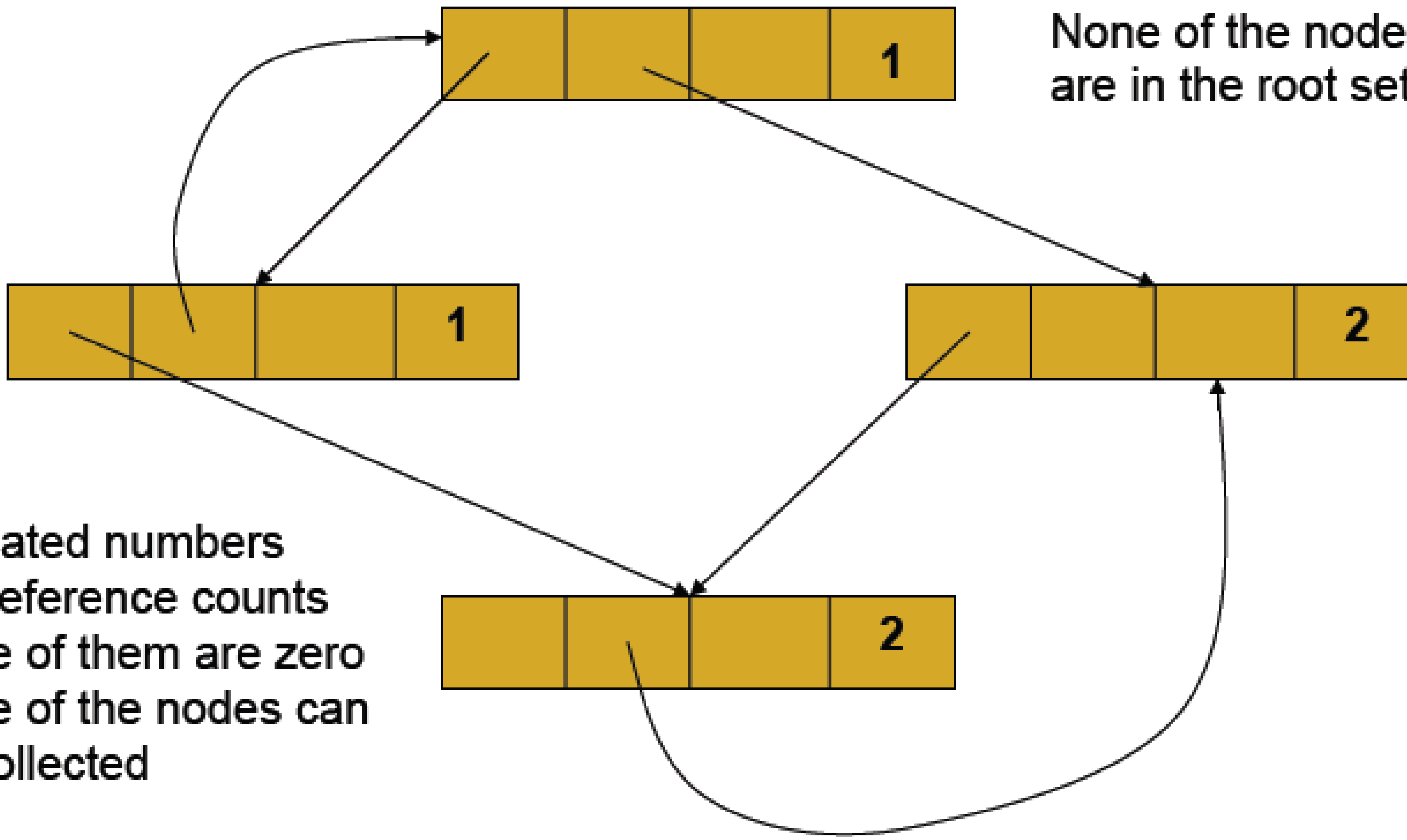
Maintaining Reference Counts

- *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 Counting GC: Disadvantages and Advantages

- 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

Mark-and-Sweep Garbage Collector

- Memory recycling steps
  - Program runs and requests memory allocations
  - GC traces and finds reachable objects
  - GC reclaims storage from unreachable objects
- Two phases
  - Marking reachable objects
  - Sweeping to reclaim storage
- Can reclaim unreachable cyclic data structures
- Stop-the-world algorithm

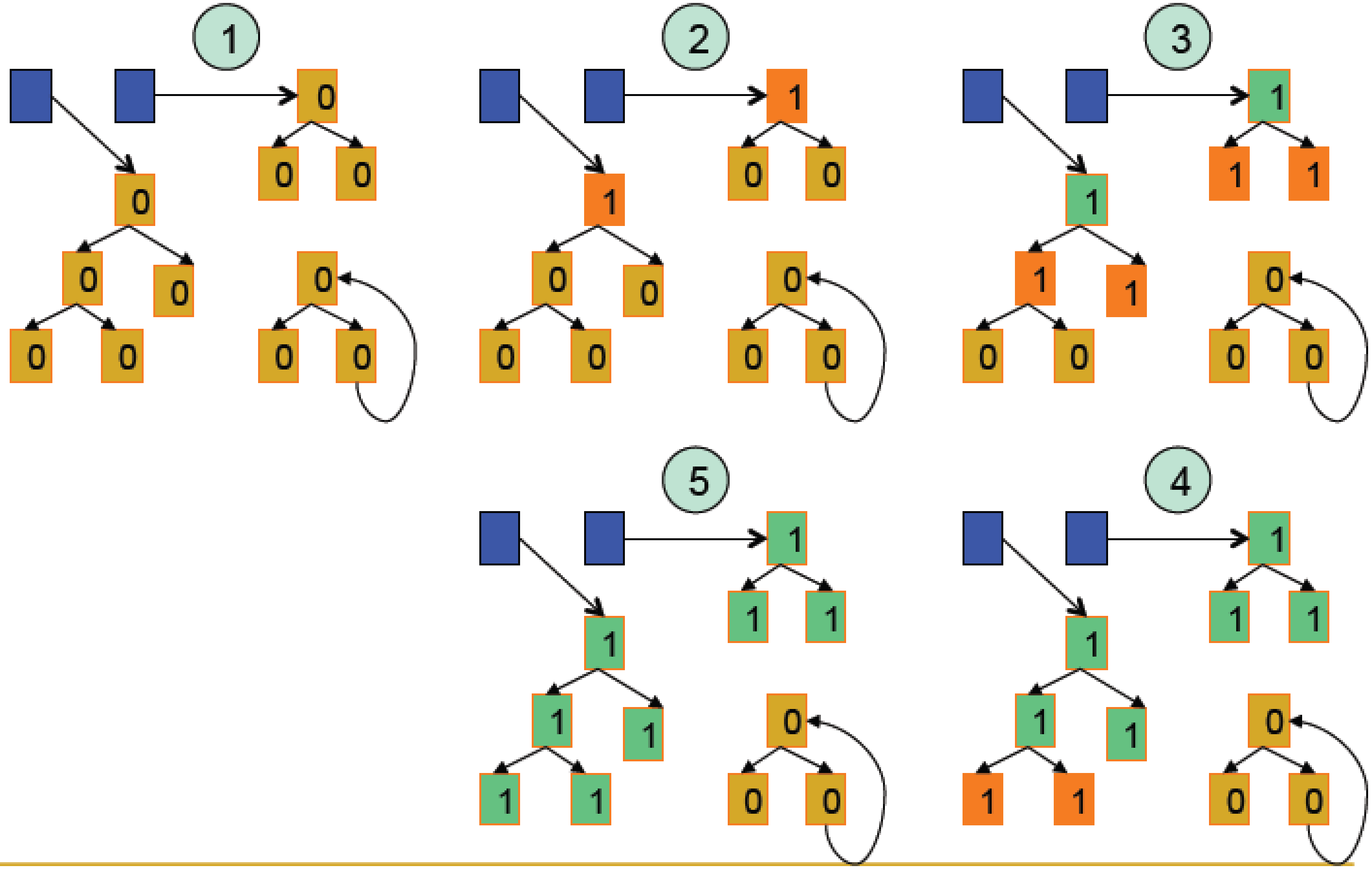
Mark-and-Sweep Algorithm - Mark

/* marking phase */

1. Start scanning from root set, mark all reachable objects (set reached-bit = 1), place them on the list Unscanned
2. while (Unscanned $\neq \Phi$) do

```
{ object o = delete(Unscanned);
  for (each object o1 referenced in o) do
     { if (reached-bit(o1) == 0)
          { reached-bit(o1) = 1; place o1 on Unscanned;}
     }
}
```

Mark-and-Sweep GC Example - Mark



Mark-and-Sweep Algorithm - Sweep

- /* Sweeping phase, each object in the heap is inspected only once */

3. Free = Φ;

```
for (each object o in the heap) do
   { if (reached-bit(o) == 0)    add(Free, o);
     else reached-bit(o) = 0;
   }
```

Mark-and-Sweep GC Example - Sweep