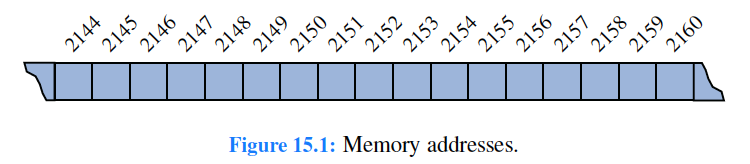
# Ch 15 Memory Management and B-Trees

# 15.1 Memory Management (p.688)

Computer memory is ***organized into a sequence of words***, each of which typically consists of 4, 8, or 16 bytes. These memory words are ***numbered from 0 to N −1***, where N is nb of memory words available to computer. The number associated with each memory word is known as its ***memory address***.



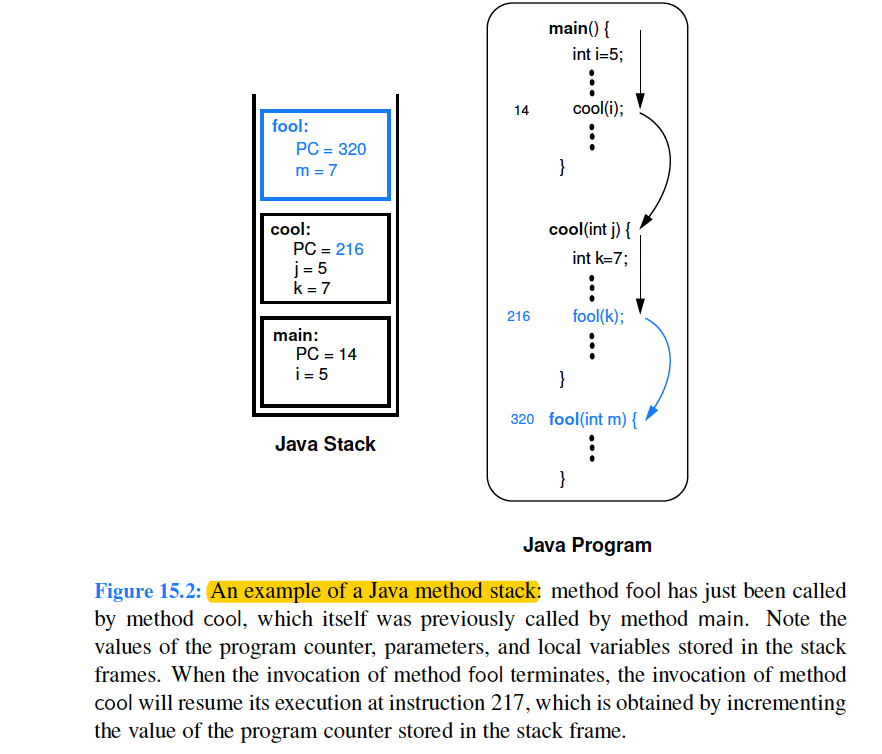
Computer’s memory must be managed so as to determine what data is stored in what memory cells.

## 15.1.1 Stacks in the JVM

A Java program is typically ***compiled into sequence of byte codes*** that serve as “machine” instructions for a well-defined model—JVM. By ***compiling Java code*** into the ***JVM byte codes***, rather than the machine language of a specific CPU, Java program ***can be run on any computer*** that has a program that can emulate the JVM.

A ***running Java program*** (a running Java thread) has a ***private stack***, called ***Java stack***, which is ***used*** to ***keep track of local variables*** and other important info on methods *as they are invoked during execution*.

During execution of Java program, ***JVM maintains stack whose elements are descriptors of currently active*** (nonterminated) invocations of methods. These descriptors are called ***frames***. A frame for some invocation of method “fool” ***stores current values of local variables*** and ***parameters*** of method fool, as well as information on method “cool” that called fool and on what needs to be returned to method “cool”.



## 15.1.2 Allocating Space in the Memory Heap

### Dynamic Memory Allocation

*Memory* for object *can be allocated dynamically during method’s execution*: by having method utilize **new** operator built into Java.

Ex: Java statement creates array of int whose size given by value of var k:

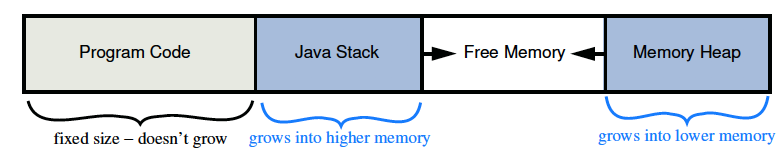
int[ ] items = new int[k];

Size of array above is known only at runtime.

Moreover, array may continue to exist even after method that created it terminates. Thus, memory for this array cannot be allocated on Java stack.

### The Memory Heap

Java uses memory from another area of storage—memory heap (not confused with “heap” data structure in ch9). We illustrate this memory area JVM in Figure 15.3. The storage available in memory heap is *divided into blocks*, which are contiguous array-like “chunks” of memory that may be of variable or fixed sizes.



### Memory allocation Algorithm

JVM definition *requires memory heap to be able to quickly allocate memory* for new objects, but it *doesn’t specify algorithm* that should be used to do this.

One method: ***keep contiguous “holes” of available free memory*** in a linked list, called *free list*. The *links joining these holes* are stored inside holes themselves, since their memory is not being used.

As memory is allocated and deallocated, collection of holes in free lists changes, with unused memory being separated into disjoint holes divided by blocks of used memory. This separation of unused memory into separate holes is known as ***fragmentation***. Problem is that it becomes *more difficult to find large continuous chunks of memory*, when needed, even though an equivalent amount of memory may be unused (yet fragmented).

Two kinds of fragmentation can occur:

***Internal fragmentation*** occurs when portion of an allocated memory block is unused. Ex: program may request array of size 1000, but only use first 100 cells of this array. A *runtime environment can’t do much to reduce internal fragmentation*.

***External fragmentation*** occurs when there is significant amount of unused memory between several contiguous blocks of allocated memory. Since *runtime environment has control over where to allocate memory when it is requested* (ex: when new keyword is used in Java), runtime environment should allocate memory in a way to try to reduce external fragmentation.

The ***best-fit algorithm*** *searches entire free list to find the hole whose size is closest to the amount of memory being requested*.

The ***first-fit algorithm*** *searches from beginning of the free list for the first hole that is large enough*. The ***first-fit algorithm*** *is fast*, but it tends to *produce a lot of external fragmentation at the front of the free list*, which *slows down future searches*.

The ***next-fit algorithm*** *searches the free list for the first hole that is large enough*, but it begins its search from where it left off previously, *viewing the free list as a circularly linked list*. The ***next-fit algorithm*** *spreads fragmentation more evenly* throughout the memory heap, thus *keeping search times low*. This spreading *makes it more difficult to allocate large blocks*.

The ***worst-fit algorithm*** *searches the free list to find the largest hole of available memory*, which might be done faster than search of entire free list if this list were maintained as a priority queue**.** The ***worst-fit algorithm*** attempts to avoid issue (more difficult to allocate large blocks) by *keeping contiguous sections of free memory as large as possible.*

## 15.1.3 Garbage Collection

*Memory* for objects is *allocated* from *memory heap*. *Space for instance variables* of running Java program are *placed in its method stacks*, one for each running thread. Since *instance variables* in method stack *can refer to objects in the memory heap*, ***all variables and objects in method stacks*** of running threads are called ***root objects***. All those *objects that can be reached by following object references that start from a root object* are called ***live objects***. The live objects are ***active objects currently being used by the running program***; these objects should not be deallocated.

*JVM may notice* that *available space* in *memory heap* is becoming *scarce*. At such times, *JVM can elect to reclaim space that is being used for objects that are no longer live*, and *return reclaimed memory to the free list*. This reclamation process is known as ***garbage collection***.

### The Mark-Sweep Algorithm

In mark-sweep garbage collection algorithm, we ***associate a “mark” bit*** *with each* ***object*** *that* ***identifies*** *whether that* ***object is live****.* *When* ***garbage collection is needed*,** we ***suspend*** *all other* ***activity*** and ***clear mark bits*** *of all objects currently allocated in memory heap*. We ***trace through Java stacks*** of the currently running threads and we ***mark all root objects*** in these stacks *as “****live****.”* We *must determine all other live objects*— the ones that are reachable from the root objects.

*To do this efficiently*, we can perform a ***depth-first search*** *on directed graph that is defined by objects referencing other objects*. In this case, *each object in memory heap* is viewed as a ***vertex*** in a directed graph, and the *reference from one object to another* is viewed as a ***directed edge***. *By performing a directed DFS* from each root object, we can correctly *identify and mark each live object*. This process is known as the ***“mark” phase***.

Once process completed, we ***scan*** *through* ***memory heap*** and ***reclaim*** *any* ***space*** that is being ***used*** *for an* ***object*** *that has* ***not*** *been* ***marked***.

Option: *merge all allocated space* in memory heap *into single block* so eliminating external fragmentation for time being. This scanning and reclamation process is known as the “***sweep” phase***, and when it completes, we resume running the suspended program.

***Mark-sweep garbage collection algorithm will reclaim unused space in time proportional to number of live objects and their references plus size of memory heap.***

### Performing DFS In-Place

Mark-sweep algorithm *correctly reclaims unused space in memory heap*, but there is an *important issue* *during the mark phase*. Since we’re *reclaiming memory space when available memory is insufficient*, we must *make sure not to use extra space during garbage collection itself*.

Trouble is that **DFS algorithm, in recursive way, *can use space proportional to number of vertices in graph****.* For garbage collection, *vertices in our graph are objects in memory heap == don’t have this much memory to use*. Only ***alternative*** is to ***find*** a ***way*** to ***perform*** ***DFS in-place rather than recursively***.

***Main idea for performing DFS in-place*** is to ***simulate*** ***recursion stack*** using the edges of the graph. When we traverse an edge from a visited vertex v to a new vertex w, we change edge (v,w) stored in v’s adjacency list to point back to v’s parent in the DFS tree. When we return back to v, we can now switch the edge we modified to point back to w.

We need to have some way of identifying which edge we need to change back: *number references* going out of v as 1, 2, and so on, and *store*, in addition to the mark bit, a *count identifier* that tells us which edges we have modified.

Using a count identifier *requires an extra word of storage per object*. This extra word *can be avoided* in some implementation. Ex: many implementations of JVM represent an object as composition of a reference with a type identifier and as a reference to other objects or data fields for this object. Since the type reference is always supposed to be the first element of the composition in such implementations we can use this reference to “mark” the edge we changed when leaving an object v and going to some object w. We simply swap reference at v that refers to type of v with reference at v that refers to w. When we return to v, we can quickly identify the edge (v,w) we changed, because it will be the first reference in the composition for v, and the position of the reference to v’s type will tell us the place where this edge belongs in v’s adjacency list.

# 15.2 Memory Hierarchies and Caching (p.695)

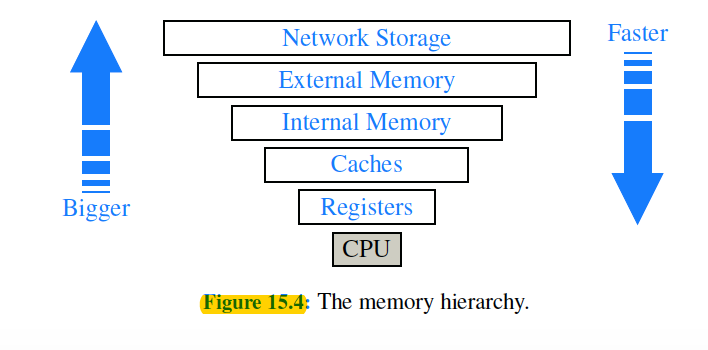
## 15.2.1 Memory Systems

To accommodate large data sets, computers have hierarchy of different kinds of memories, which vary in size and distance from CPU. ***Closest to CPU are internal registers*** *that CPU itself uses*. Access to such locations is very fast, but there are relatively few such locations.

At ***second level*** in hierarchy are one or more ***memory caches***. This memory is *considerably larger than register set of a CPU*, but *accessing it takes longer*.

At ***third level*** in hierarchy is the ***internal memory***, which is also *known as main memory or core memory*. The internal memory is *considerably larger than cache memory*, but also *requires more time to access*.

***Another level*** in hierarchy is ***external memory***, which usually consists of disks, CD drives, DVD drives, and/or tapes. This memory is *very large, but it is also very slow*.



## 15.2.2 Caching Strategies

The ***significance of memory hierarchy*** on the performance of a program ***depends*** upon the ***size of the problem*** and ***physical characteristics of computer system***. The *bottleneck occurs between two levels of memory hierarchy*—the one that can hold all data items and the level just below that one. For a problem that can fit entirely in main memory, the two most important levels are the cache memory and the internal memory.

Access times for internal memory can be as much as 10 to 100 times longer than those for cache memory. It is d*esirable to be able to perform most memory accesses in cache memory*. For a problem that does not fit entirely in main memory, on the other hand, the two most important levels are the internal memory and the external memory.

*Most algorithms are not designed with memory hierarchy in mind*, in spite of the great variance between access times for the different levels. All of algorithm analyses described in this book have assumed that all memory accesses are equal. One justification for this assumption is that ***it is often necessary to assume that all memory accesses take the same amount of time***, since specific device-dependent information about memory sizes is often hard to come by. In fact, information about memory size may be difficult to get.

### Caching and Blocking

Another justification for memory-access equality assumption is that ***operating system designers have developed general mechanisms that allow most memory accesses to be fast***. These mechanisms are ***based on two important locality-of-reference properties*** that most software possesses:

• **Temporal locality**: *If program accesses a certain memory location, then there is increased likelihood that it accesses that same location again in near future.*

Ex: common to use value of a counter variable in several diff expressions, including one to increment counter’s value. A common adage among computer architects is that a program spends 90% of its time in 10% of its code.

• **Spatial locality**: *If program accesses a certain memory location, then there is increased likelihood that it soon accesses other locations that are near this one*.

Ex: a program using an array may be likely to access the locations of this array in a sequential or near-sequential manner.

Temporal and spatial localities have ***given rise to two fundamental design choices*** for multilevel computer memory systems:

* ***virtual memory***: concept consists of *providing an address space as large as capacity of the secondary-level memory*, and of *transferring data located in secondary level into the primary level*.

Virtual memory does *not limit* programmer to the constraint of the *internal memory size*. The *concept of bringing data into primary memory* is called ***caching***, and it is *motivated by temporal locality*. By bringing data into primary memory, we are hoping that it will be accessed again soon, and we will be able to respond quickly to all the requests for this data that come in the near future.

* ***motivated by spatial locality:*** if data stored at a secondary-level memory location ℓ is accessed, then we bring into primary-level memory a large block of contiguous locations that include the location ℓ. This concept is known as ***blocking***, and it is *motivated by expectation that other secondary-level memory locations close to ℓ will soon be accessed*. In *interface between cache memory and internal memory*, such blocks are often called ***cache lines***, and *in the interface between internal memory and external memory*, such blocks are often called ***pages***.

When implemented with caching and blocking, virtual memory often allows us to perceive secondary-level memory as being faster than it really is. There is still a problem. ***Primary-level memory is much smaller than secondary level memory***. Moreover, because *memory systems use blocking*, any program of substance *will likely reach a point where it requests data from secondary-level memory*, but the primary memory is already full of blocks.

### Caching in Web Browsers

*To exploit temporal locality of reference*, it is often advantageous to *store copies of Web pages in cache memory*, so these pages can be *quickly retrieved when* *requested again*.

***This creates a two-level memory hierarchy***, with *cache serving as smaller, quicker internal memory*, and *network being the external memory*.

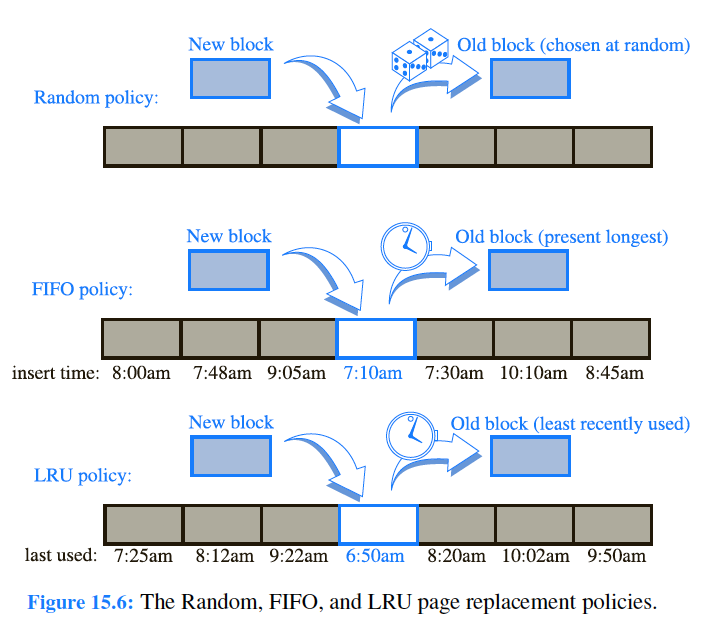
In particular, suppose we have a cache memory that has m “slots” that *can contain Web pages. We* assume that a Web page can be placed in any slot of the cache. This is known as a ***fully associative cache***. As a browser executes, it requests different Web pages. *Each time the browser requests such a Web page p, the browser determines if p is unchanged and currently contained in the cache*.

If p is contained in the cache, then browser satisfies the request using the cached copy. If p is not in the cache, however, page for p is requested over the Internet and transferred into the cache. If one of the m slots in the cache is available, then the browser assigns p to one of the empty slots. But if all the m cells of the cache are occupied, then the computer must determine which previously viewed Web page to evict before bringing in p to take its place.

### Page Replacement Algorithms

Some of the better-known page replacement policies include:

* ***First-in, first-out (FIFO***): Evict the page that has been in cache the longest, the page that was transferred to the cache furthest in the past.
* ***Least recently used (LRU***): Evict the page whose last request occurred furthest in the past.
* ***Random***: Choose a page at random to evict from the cache. Easiest policies to implement, only requires a random/pseudorandom number generator. The overhead involved in implementing this policy is an O(1) additional amount of work per page replacement.



***FIFO strategy*** is simple to implement, *only requires a queue Q to store references to the pages in the cache*. *Pages are enqueued in Q when they are referenced* by a browser, and then are brought into the cache. When a *page needs to be evicted*, the computer simply performs a *dequeue operation* on Q to determine which page to evict. This policy also requires O(1) additional work per page replacement. Also, the FIFO policy incurs no additional overhead for page requests.

***LRU strategy*** *takes advantage of temporal locality as much as possible*, *by always evicting the page that was least-recently used*. From a policy point of view, this is an excellent approach, but *it is costly from an implementation point of view*. Its way of optimizing temporal and spatial locality is fairly costly. Implementing LRU strategy requires the use of an adaptable priority queue Q that supports updating the priority of existing pages. If Q is implemented with a sorted sequence based on a linked list, then the overhead for each page request and page replacement is O(1). When we insert a page in Q or update its key, the page is assigned the highest key in Q and is placed at the end of the list, which can also be done in O(1) time. Even though the LRU strategy has constant-time overhead.

Since these different page replacement policies have different trade-offs between implementation difficulty and degree to which they seem to take advantage of localities == which one is the best using comparative analysis.

From a ***worst-case point of view***, FIFO and LRU strategies have fairly unattractive competitive behavior. This worst-case analysis is a l*ittle too pessimistic* for it focuses on each protocol’s behavior for one bad sequence of page requests. *An ideal analysis would be to compare these methods over all possible page-request sequences*.

# 15.3 External Searching and B-Trees (p.701)

## 15.3.1 (a,b) Trees

## 15.3.2 B-Trees

# 15.4 External-Memory Sorting (p.705)

## 15.4.1 Multiway Merging