Memory Management

Dynamic Memory Allocation

- Lots of things need memory at runtime:
  - Activation records
  - Objects
  - Explicit allocations: **`new`**, **`malloc`**, etc.
  - Implicit allocations: strings, file buffers, arrays with dynamically varying size, etc.
- Language systems provide an important hidden player: runtime memory management

Outline

- 14.2 Memory model using Java arrays
- 14.3 Stacks
- 14.4 Heaps
- 14.5 Current heap links
- 14.5 Garbage collection

Memory Model

- For now, assume that the OS grants each running program one or more fixed-size regions of memory for dynamic allocation
- We will model these regions as Java arrays
  - To see examples of memory management code
  - And, for practice with Java

Declaring An Array

- A Java array declaration:

```
int[] a = null;
```

- Array types are reference types—an array is really an object, with a little special syntax
- The variable **`a`** above is initialized to **`null`**
- It can hold a reference to an array of **`int`** values, but does not yet

Creating An Array

- Use **new** to create an array object:

```
int[] a = null;
a = new int[100];
```

- We could have done it with one declaration statement, like this:

```
int[] a = new int[100];
```

Using An Array

```
int i = 0;
while (i<a.length) {
  a[i] = 5;
  i++;
}
```

- Use **`a[i]`** to refer to an element (as lvalue or rvalue): **`a`** is an array reference expression and **`i`** is an **`int`** expression
- Use **`a.length`** to access length
- Array indexes are 0..(**`a.length-1`**)

Memory Managers In Java

```
public class MemoryManager {
  private int[] memory;

  /**
   * MemoryManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public MemoryManager(int[] initialMemory) {
    memory = initialMemory;
  }
  …
}
```

We will show Java implementations this way. The **`initialMemory`** array is the memory region provided by the operating system.

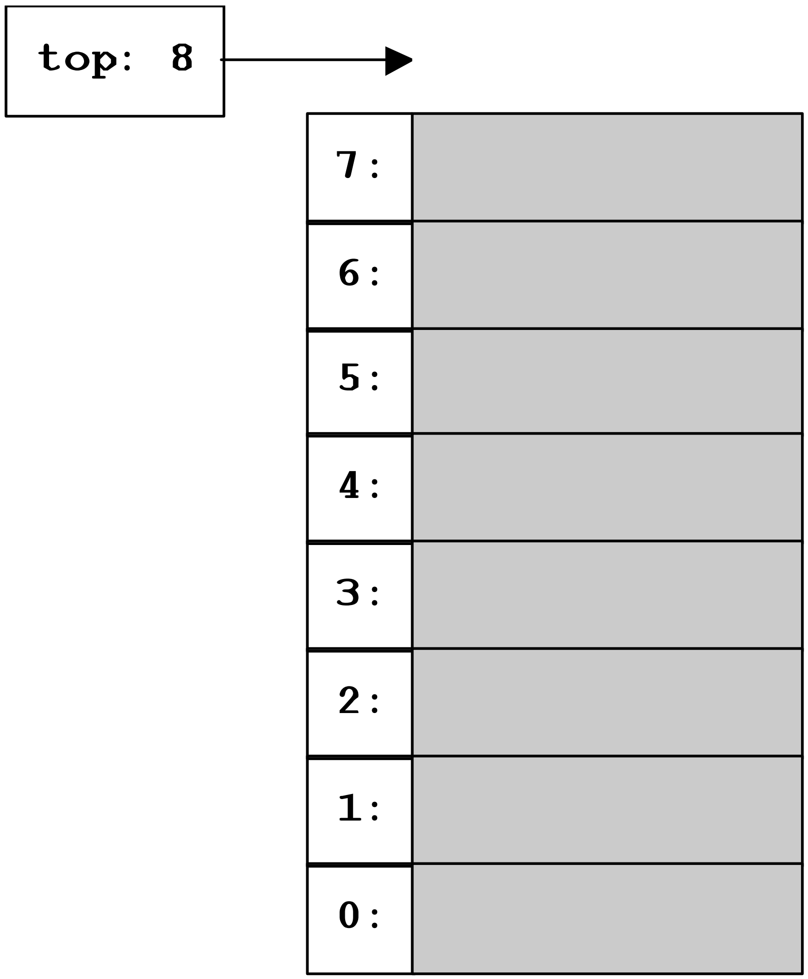
Outline

- 14.2 Memory model using Java arrays
- 14.3 Stacks
- 14.4 Heaps
- 14.5 Current heap links
- 14.5 Garbage collection

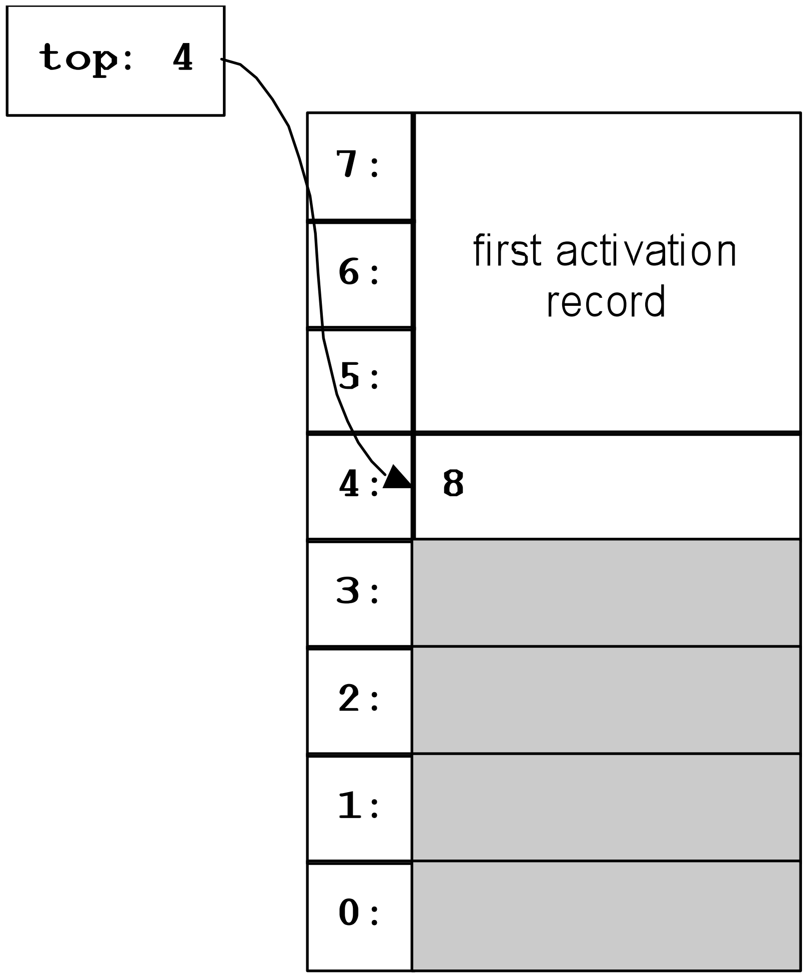
Stacks Of Activation Records

- For almost all languages, activation records must be allocated dynamically
- For many, it suffices to allocate on call and deallocate on return
- This produces a stack of activation records: push on call, pop on return
- A simple memory management problem

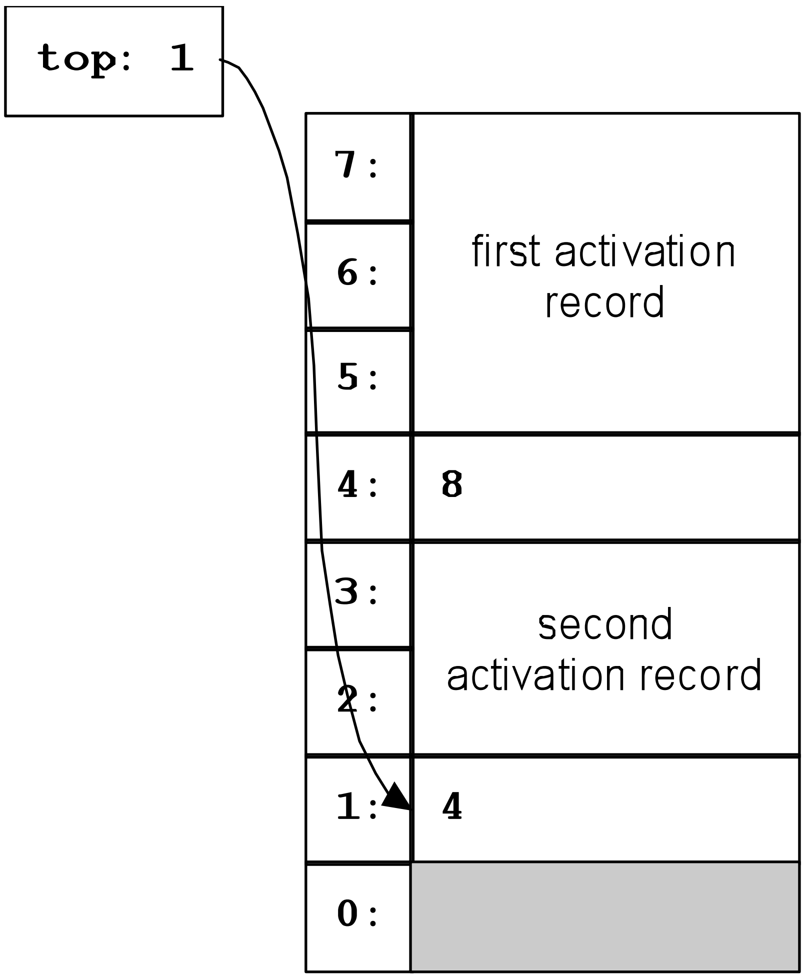
A Stack Illustration



An empty stack of 8 words. The stack will grow down, from high addresses to lower addresses. A reserved memory location (perhaps a register) records the address of the lowest allocated word.



The program calls **`m.push(3)`**, which returns 5: the address of the first of the 3 words allocated for an activation record. Memory management uses an extra word to record the previous value of **`top`**.



The program calls **`m.push(2)`**, which returns 2: the address of the first of the 2 words allocated for an activation record. The stack is now full—there is not room even for **`m.push(1)`**.

For **`m.pop()`**, just do

`top = memory[top]`

to return to previous configuration.

A Java Stack Implementation

```
public class StackManager {
  private int[] memory; // the memory we manage
  private int top; // index of top stack block

  /**
   * StackManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public StackManager(int[] initialMemory) {
    memory = initialMemory;
    top = memory.length;
  }
```

```
/**
 * Allocate a block and return its address.
 * @param requestSize int size of block, > 0
 * @return block address
 * @throws StackOverflowError if out of stack space
 */
public int push(int requestSize) {
  int oldtop = top;
  top -= (requestSize+1); // extra word for oldtop
  if (top<0) throw new StackOverflowError();
  memory[top] = oldtop;
  return top+1;
}
```

The **`throw`** statement and exception handling are introduced in Chapter 17.

```
  /**
   * Pop the top stack frame.  This works only if the
   * stack is not empty.
   */
  public void pop() {
    top = memory[top];
  }
}
```

Outline

- 14.2 Memory model using Java arrays
- 14.3 Stacks
- 14.4 Heaps
- 14.5 Current heap links
- 14.5 Garbage collection

The Heap Problem

- Stack order makes implementation easy
- Not always possible: what if allocations and deallocations can come in any order?
- A *heap* is a pool of blocks of memory, with an interface for unordered runtime memory allocation and deallocation
- There are many mechanisms for this…

First Fit

- A linked list of free blocks, initially containing one big free block
- To allocate:
  - Search free list for first adequate block
  - If there is extra space in the block, return the unused portion at the upper end to the free list
  - Allocate requested portion (at the lower end)
- To free, just add to the front of the free list

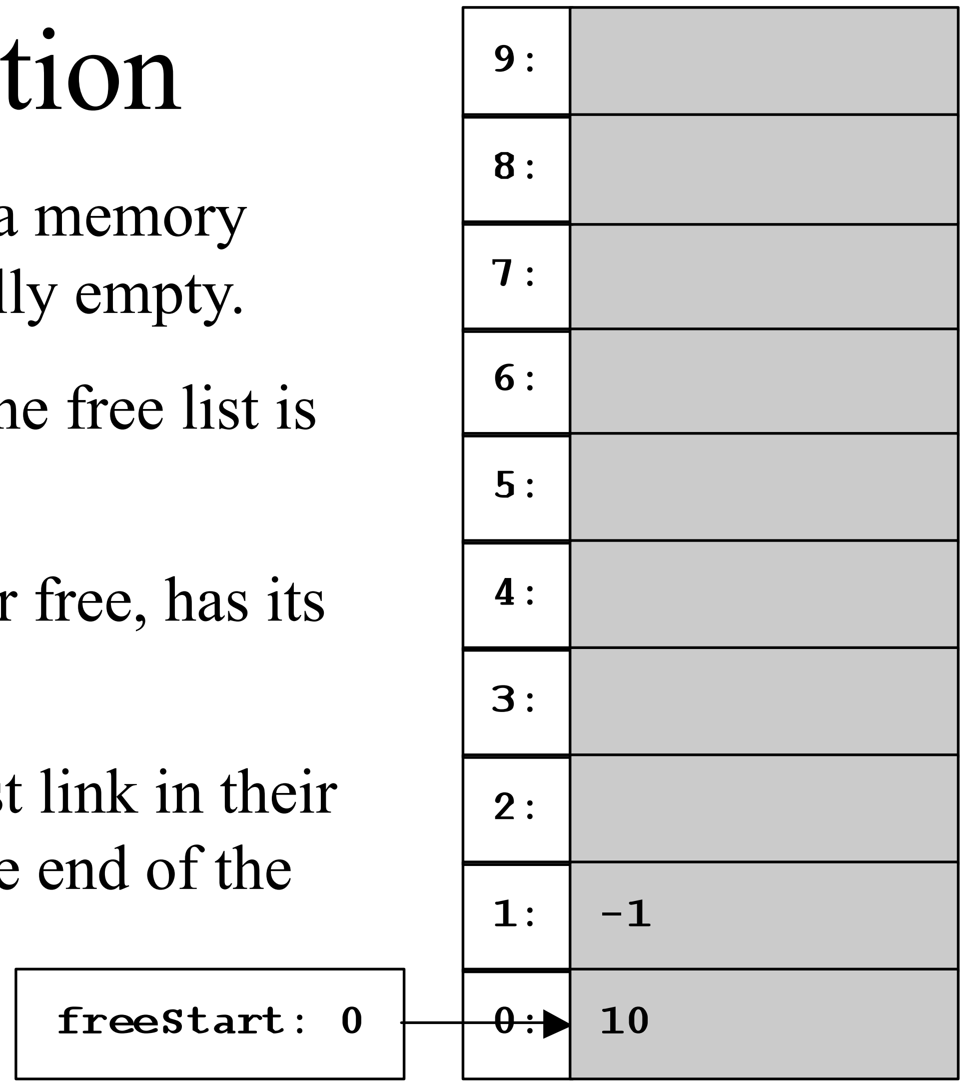
Heap Illustration

A heap manager **`m`** with a memory array of 10 words, initially empty.

The link to the head of the free list is held in **`freeStart`**.

Every block, allocated or free, has its length in its first word.

Free blocks have free-list link in their second word, or –1 at the end of the free list.

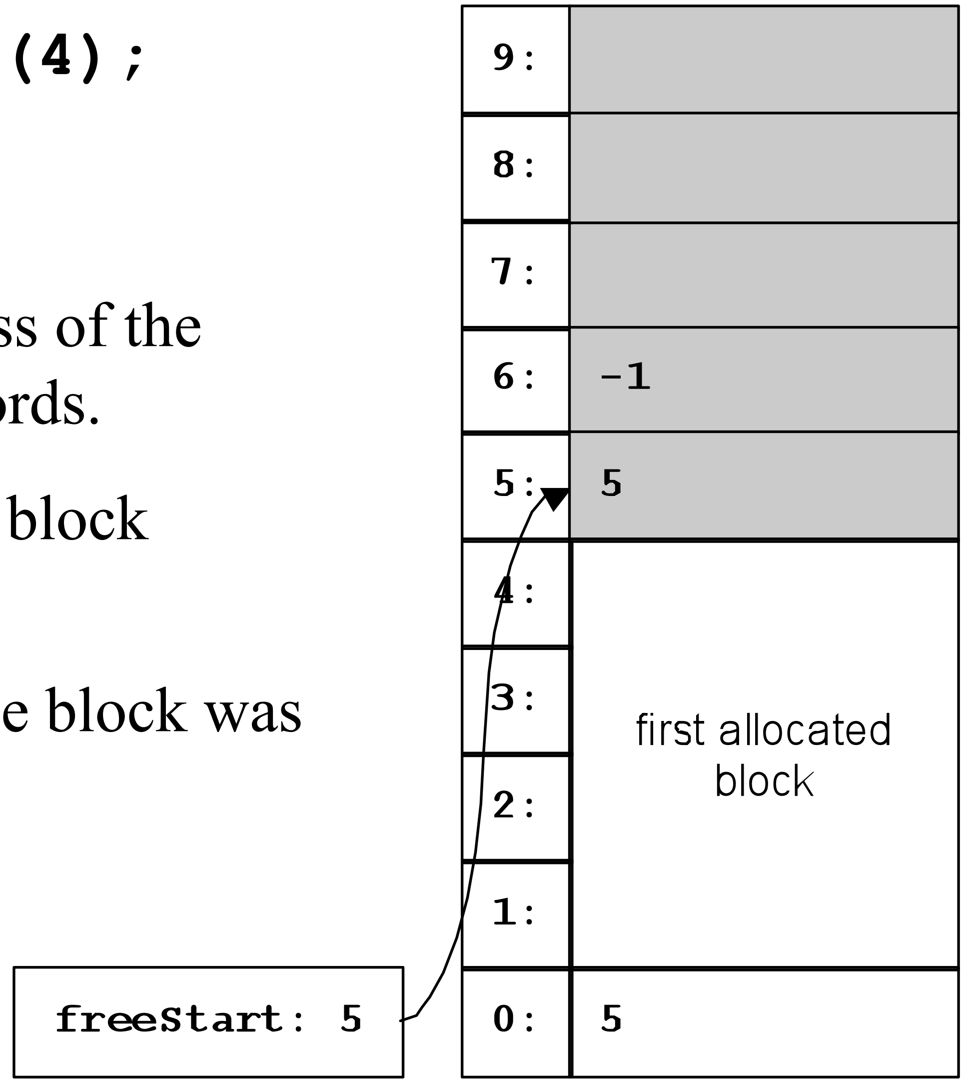


```
p1=m.allocate(4);
```

`p1` will be 1—the address of the first of four allocated words.

An extra word holds the block length.

Remainder of the big free block was returned to the free list.

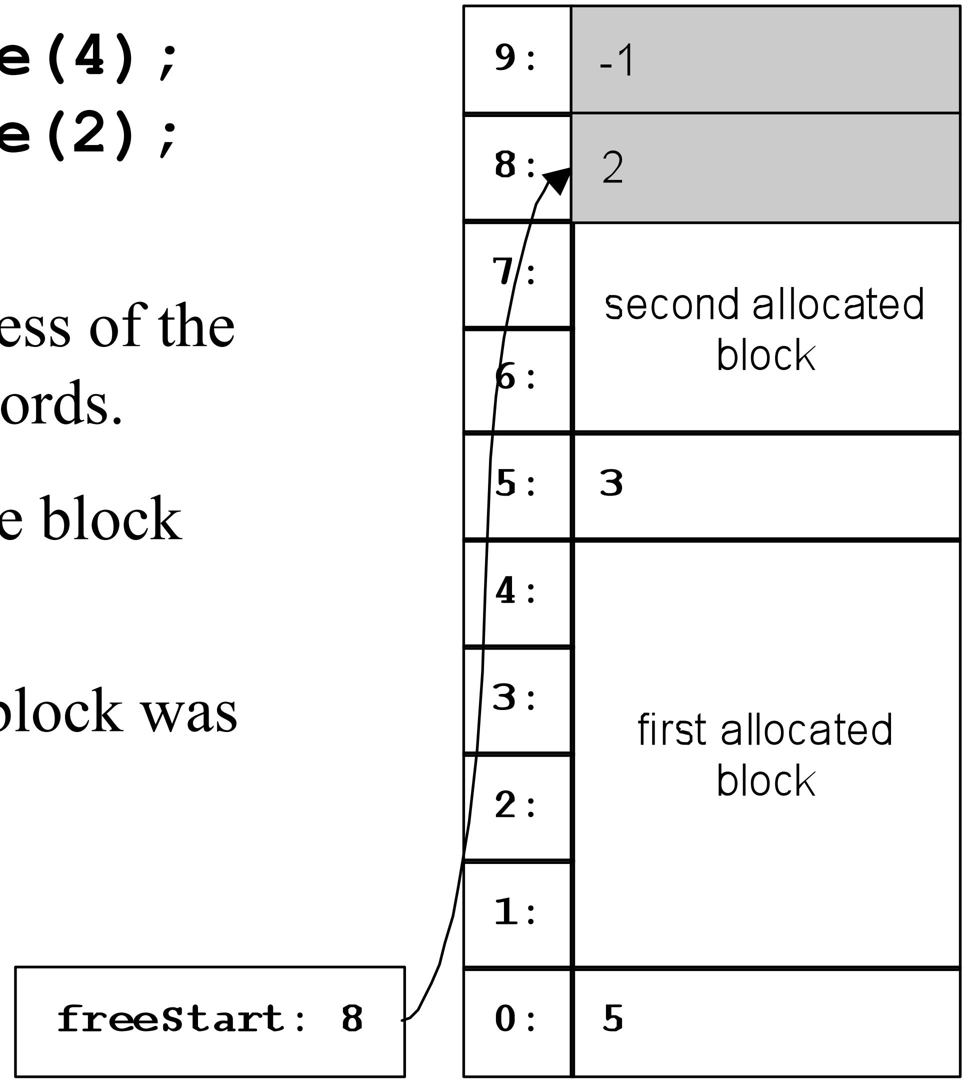


```
p1=m.allocate(4);
p2=m.allocate(2);
```

`p2` will be 6—the address of the first of two allocated words.

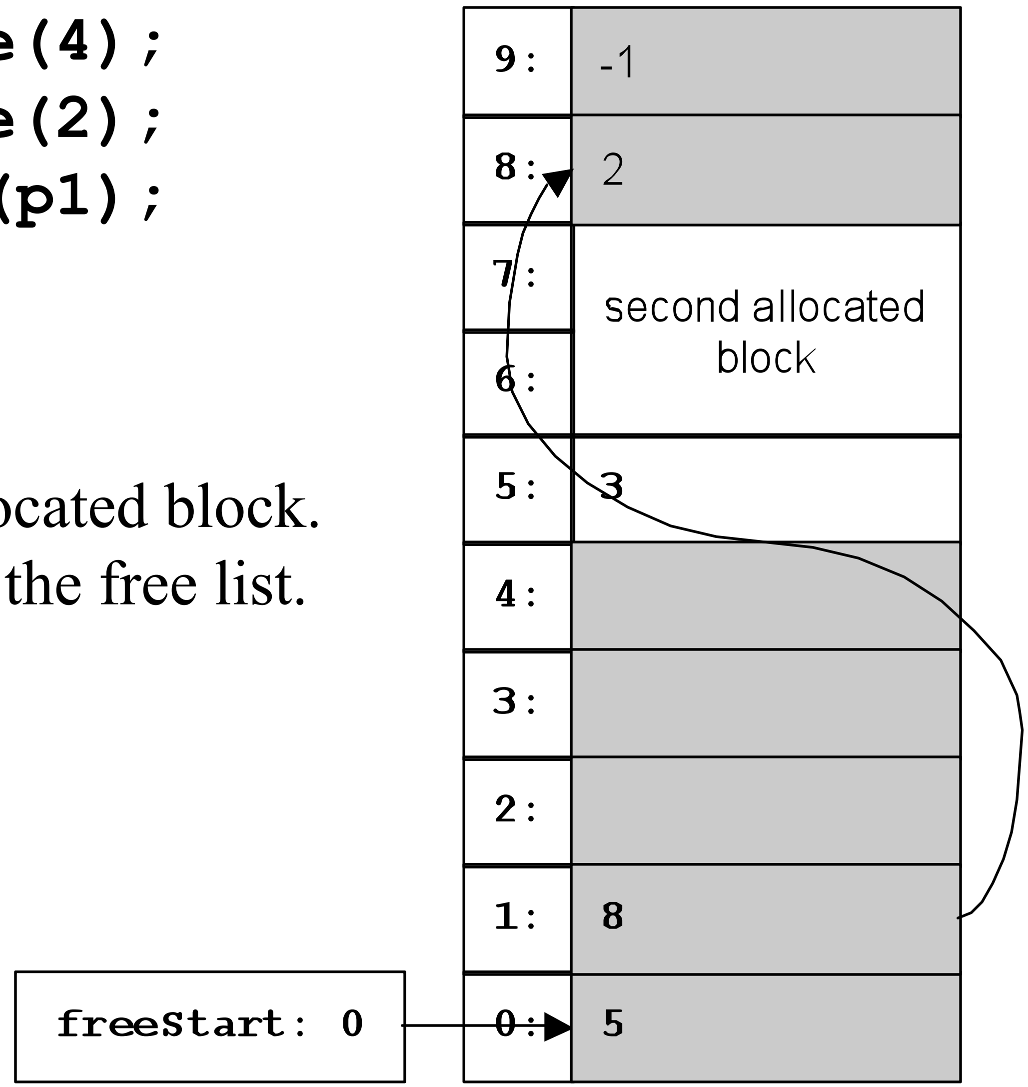
An extra word holds the block length.

Remainder of the free block was returned to the free list.



```
p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);
```

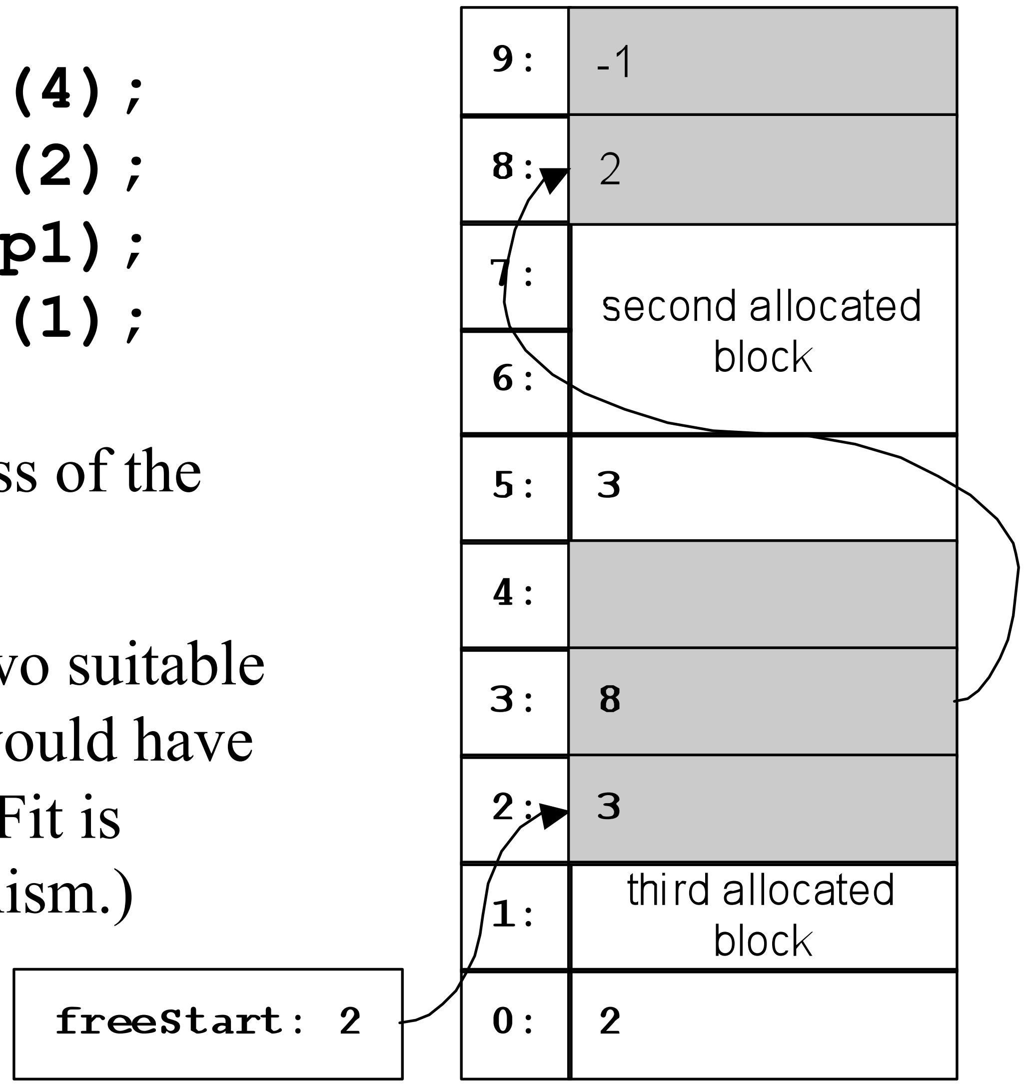
Deallocates the first allocated block.
It returns to the head of the free list.



```
p1=m.allocate(4);
p2=m.allocate(2);
m.deallocate(p1);
p3=m.allocate(1);
```

p3 will be 1—the address of the allocated word.

Notice that there were two suitable blocks. The other one would have been an exact fit. (Best Fit is another possible mechanism.)



A Java Heap Implementation

```
public class HeapManager {
  static private final int NULL = -1; // null link
  public int[] memory; // the memory we manage
  private int freeStart; // start of the free list

  /**
   * HeapManager constructor.
   * @param initialMemory int[] of memory to manage
   */
  public HeapManager(int[] initialMemory) {
    memory = initialMemory;
    memory[0] = memory.length; // one big free block
    memory[1] = NULL; // free list ends with it
    freeStart = 0; // free list starts with it
  }
```

```
/**
 * Allocate a block and return its address.
 * @param requestSize int size of block, > 0
 * @return block address
 * @throws OutOfMemoryError if no block big enough
 */
public int allocate(int requestSize) {
  int size = requestSize + 1; // size with header

  // Do first-fit search: linear search of the free
  // list for the first block of sufficient size.
  int p = freeStart; // head of free list
  int lag = NULL;
  while (p!=NULL && memory[p]<size) {
    lag = p; // lag is previous p
    p = memory[p+1]; // link to next block
  }
  if (p==NULL) // no block large enough
    throw new OutOfMemoryError();
  int nextFree = memory[p+1]; // block after p
```

```
  // Now p is the index of a block of sufficient size,
  // and lag is the index of p's predecessor in the
  // free list, or NULL, and nextFree is the index of
  // p's successor in the free list, or NULL.
  // If the block has more space than we need, carve
  // out what we need from the front and return the
  // unused end part to the free list.
  int unused = memory[p]-size; // extra space
  if (unused>1) { // if more than a header's worth
    nextFree = p+size; // index of the unused piece
    memory[nextFree] = unused; // fill in size
    memory[nextFree+1] = memory[p+1]; // fill in link
    memory[p] = size; // reduce p's size accordingly
  }

  // Link out the block we are allocating and done.
  if (lag==NULL) freeStart = nextFree;
  else memory[lag+1] = nextFree;
  return p+1; // index of useable word (after header)
}
```

```
  /**
   * Deallocate an allocated block.  This works only if
   * the block address is one that was returned by
   * allocate and has not yet been deallocated.
   * @param address int address of the block
   */
  public void deallocate(int address) {
    int addr = address-1;
    memory[addr+1] = freeStart;
    freeStart = addr;
  }
}
```

A Problem

- Consider this sequence:

```
p1=m.allocate(4);
p2=m.allocate(4);
m.deallocate(p1);
m.deallocate(p2);
p3=m.allocate(7);
```

- Final **`allocate`** will fail: we are breaking up large blocks but never reversing the process
- Need to *coalesce* adjacent free blocks

A Solution

- We can implement a smarter **`deallocate`** method:
  - Maintain the free list sorted in address order
  - When freeing, look at the previous free block and the next free block
  - If adjacent, coalesce
- This is a lot more work than just returning the block to the head of the free list…

```
/**
  * Deallocate an allocated block.  This works only if
  * the block address is one that was returned by
  * allocate and has not yet been deallocated.
  * @param address int address of the block
  */
 public void deallocate(int address) {
   int addr = address-1; // real start of the block

   // Find the insertion point in the sorted free list
   // for this block.

   int p = freeStart;
   int lag = NULL;
   while (p!=NULL && p<addr) {
     lag = p;
     p = memory[p+1];
   }
```

```
// Now p is the index of the block to come after
// ours in the free list, or NULL, and lag is the
// index of the block to come before ours in the
// free list, or NULL.

// If the one to come after ours is adjacent to it,
// merge it into ours and restore the property
// described above.

if (addr+memory[addr]==p) {
  memory[addr] += memory[p]; // add its size to ours
  p = memory[p+1]; //
}
```

```
    if (lag==NULL) { // ours will be first free
      freeStart = addr;
      memory[addr+1] = p;
    }
    else if (lag+memory[lag]==addr) { // block before is
                                      // adjacent to ours
      memory[lag] += memory[addr]; // merge ours into it
      memory[lag+1] = p;
    }
    else { // neither: just a simple insertion
      memory[lag+1] = addr;
      memory[addr+1] = p;
    }
  }
```

Quick Lists

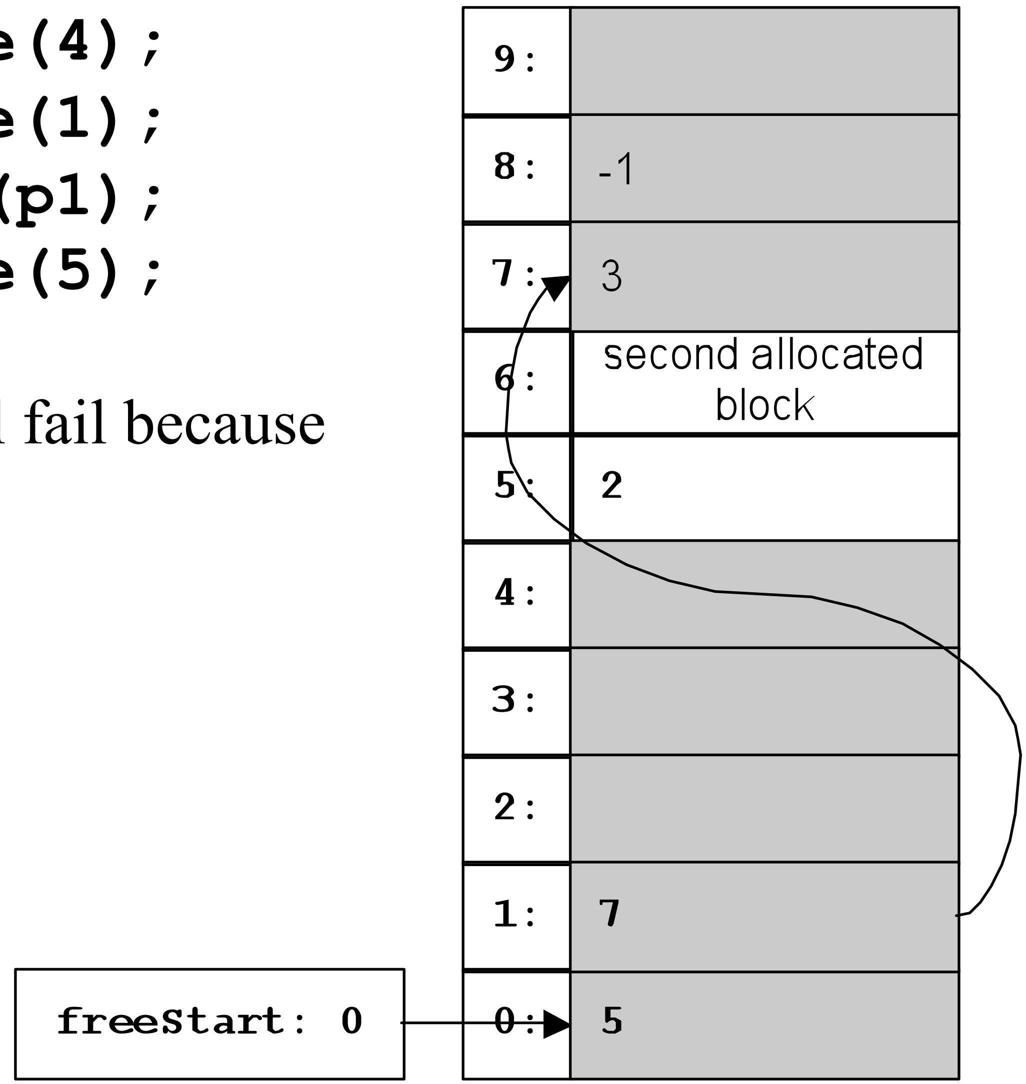
- Small blocks tend to be allocated and deallocated much more frequently
- A common optimization: keep separate free lists for popular (small) block sizes
- On these *quick lists*, blocks are one size
- *Delayed coalescing*: free blocks on quick lists are not coalesced right away (but may have to be coalesced eventually)

Fragmentation

- When free regions are separated by allocated blocks, so that it is not possible to allocate all of free memory as one block
- More generally: any time a heap manager is unable to allocate memory even though free
  - If it allocated more than requested
  - If it does not coalesce adjacent free blocks
  - And so on…

```
p1=m.allocate(4);
p2=m.allocate(1);
m.deallocate(p1);
p3=m.allocate(5);
```

The final allocation will fail because of fragmentation.



Other Heap Mechanisms

- An amazing variety
- Three major issues:
  - Placement—where to allocate a block
  - Splitting—when and how to split large blocks
  - Coalescing—when and how to recombine
- Many other refinements

Placement

- Where to allocate a block
- Our mechanism: first fit from FIFO free list
- Some mechanisms use a similar linked list of free blocks: first fit, best fit, next fit, etc.
- Some mechanisms use a more scalable data structure like a balanced binary tree

Splitting

- When and how to split large blocks
- Our mechanism: split to requested size
- Sometimes you get better results with less splitting—just allocate more than requested
- A common example: rounding up allocation size to some multiple

Coalescing

- When and how to recombine adjacent free blocks
- We saw several varieties:
  - No coalescing
  - Eager coalescing
  - Delayed coalescing (as with quick lists)

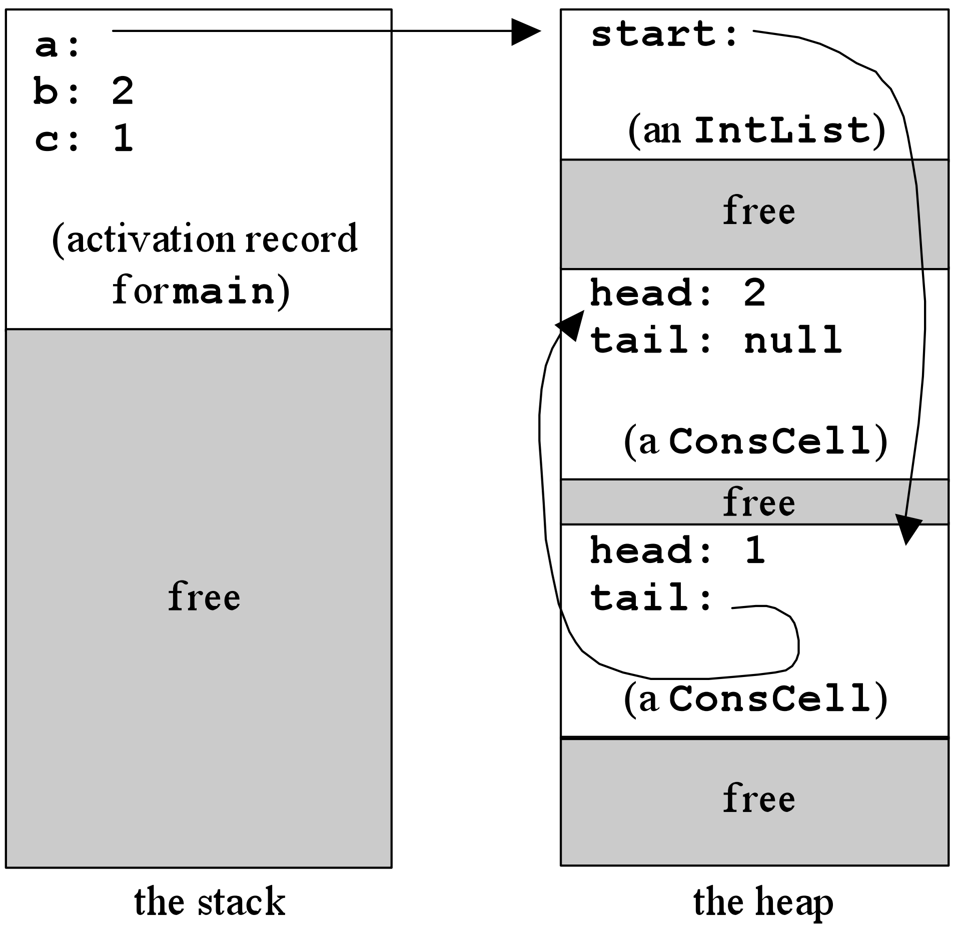
Outline

- 14.2 Memory model using Java arrays
- 14.3 Stacks
- 14.4 Heaps
- 14.5 Current heap links
- 14.5 Garbage collection

Current Heap Links

- So far, the running program is a black box: a source of allocations and deallocations
- What does the running program do with addresses allocated to it?
- Some systems track current heap links
- A *current heap link* is a memory location where a value is stored that the running program will use as a heap address

Tracing Current Heap Links




```
IntList a =
  new IntList(null);
int b = 2;
int c = 1;
a = a.cons(b);
a = a.cons(c);
```

Where are the current heap links in this picture?

To Find Current Heap Links

- Start with the *root set*: memory locations outside of the heap with links into the heap
  - Active activation records (if on the stack)
  - Static variables, etc.
- For each memory location in the set, look at the allocated block it points to, and add all the memory locations in that block
- Repeat until no new locations are found

Discarding Impossible Links

- Depending on the language and implementation, we may be able to discard some locations from the set:
  - If they do not point into allocated heap blocks
  - If they do not point *to* allocated heap blocks (Java, but not C)
  - If their dynamic type rules out use as heap links
  - If their static type rules out use as heap links (Java, but not C)

Errors In Current Heap Links

- *Exclusion errors*: a memory location that actually is a current heap link is left out
- *Unused inclusion errors*: a memory location is included, but the program never actually uses the value stored there
- *Used inclusion errors*: a memory location is included, but the program uses the value stored there as something other than a heap address—as an integer, for example

Errors Are Unavoidable

- For heap manager purposes, exclusion errors are unacceptable
- We must include a location if it *might* be used as a heap link
- This makes unused inclusion errors unavoidable
- Depending on the language, used inclusions may also be unavoidable

Used Inclusion Errors In C

- Static type and runtime value may be of no use in telling how a value will be used
- Variable **`x`** may be used either as a pointer or as an array of four characters

```
union {
  char *p;
  char tag[4];
} x;
```

Heap Compaction

- One application for current heap links
- Manager can move allocated blocks:
  - Copy the block to a new location
  - Update all links to (or into) that block
- So it can *compact* the heap, moving all allocated blocks to one end, leaving one big free block and no fragmentation

Outline

- 14.2 Memory model using Java arrays
- 14.3 Stacks
- 14.4 Heaps
- 14.5 Current heap links
- 14.5 Garbage collection

Some Common Pointer Errors

```
type
  p: ^Integer;
begin
  new(p);
  p^ := 21;
  dispose(p);
  p^ := p^ + 1
end
```

Dangling pointer: this Pascal fragment uses a pointer after the block it points to has been deallocated

```
procedure Leak;
  type
    p: ^Integer;
  begin
    new(p)
  end;
```

Memory leak: this Pascal procedure allocates a block but forgets to deallocate it

Garbage Collection

- Since so many errors are caused by improper deallocation…
- …and since it is a burden on the programmer to have to worry about it…
- …why not have the language system reclaim blocks automatically?

Three Major Approaches

- Mark and sweep
- Copying
- Reference counting

Mark And Sweep

- A mark-and-sweep collector uses current heap links in a two-stage process:
  - *Mark*: find the live heap links and mark all the heap blocks linked to by them
  - *Sweep*: make a pass over the heap and return unmarked blocks to the free pool
- Blocks are not moved, so both kinds of inclusion errors are tolerated

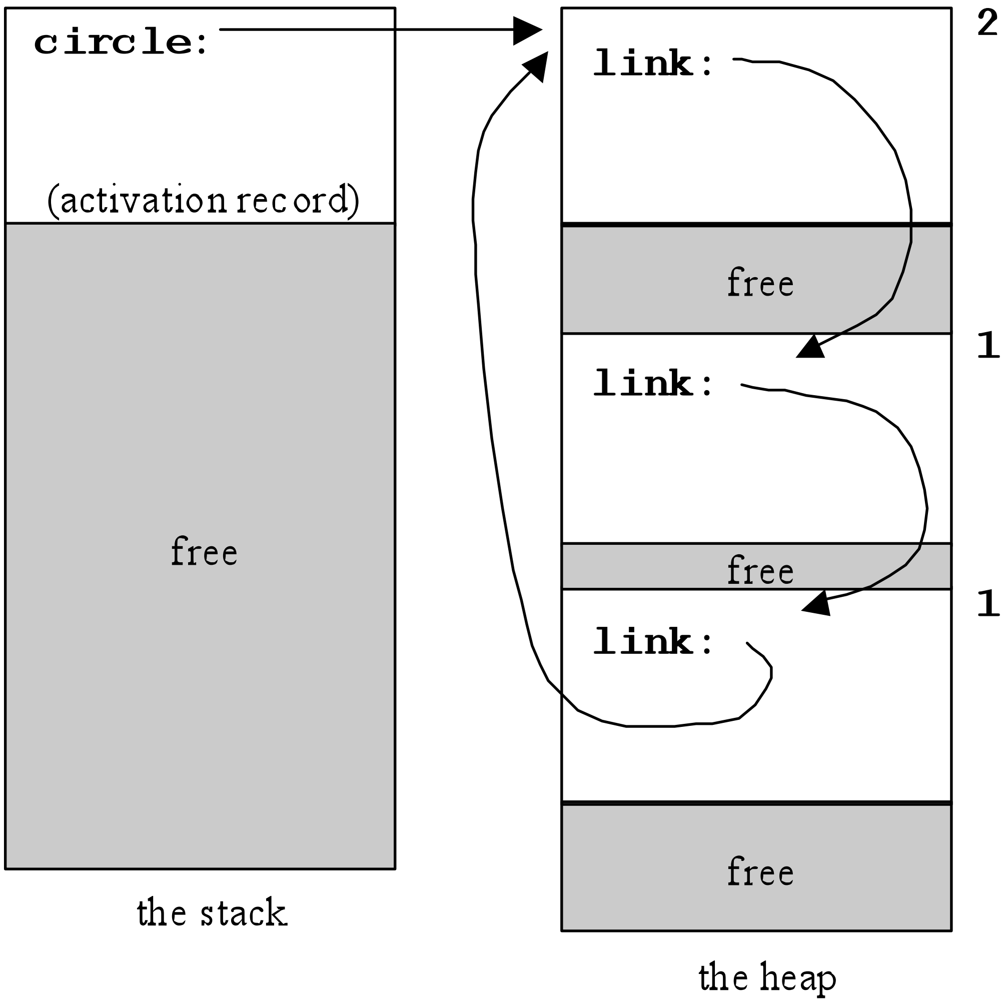
Copying Collection

- A copying collector divides memory in half, and uses only one half at a time
- When one half becomes full, find live heap links, and copy live blocks to the other half
- Compacts as it goes, so fragmentation is eliminated
- Moves blocks: cannot tolerate used inclusion errors

Reference Counting

- Each block has a counter of heap links to it
- Incremented when a heap link is copied, decremented when a heap link is discarded
- When counter goes to zero, block is garbage and can be freed
- Does not use current heap links

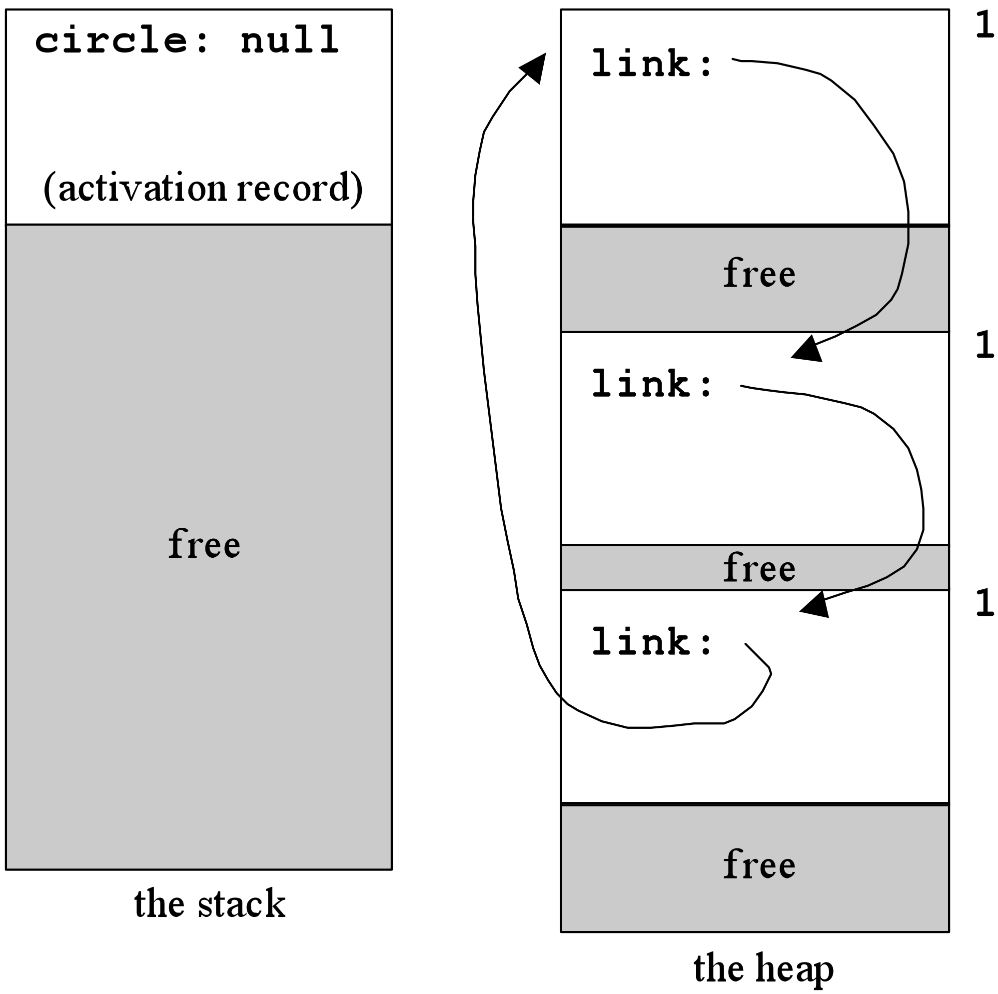
Reference Counting Problem



One problem with reference counting: it misses cycles of garbage.

Here, a circularly linked list is pointed to by **`circle`**.

Reference Counting Problem



When **`circle`** is set to null, the reference counter is decremented.

No reference counter is zero, though all blocks are garbage.

Reference Counting

- Problem with cycles of garbage
- Problem with performance generally, since the overhead of updating reference counters is high
- One advantage: naturally incremental, with no big pause while collecting

Garbage Collecting Refinements

- *Generational* collectors
  - Divide block into *generations* according to age
  - Garbage collect in younger generations more often (using previous methods)
- *Incremental* collectors
  - Collect garbage a little at a time
  - Avoid the uneven performance of ordinary mark-and-sweep and copying collectors

Garbage Collecting Languages

- Some require it: Java, ML
- Some encourage it: Ada
- Some make it difficult: C, C++
  - Even for C and C++ it is possible
  - There are libraries that replace the usual **`malloc`**/**`free`** with a garbage-collecting manager

Trends

- An old idea whose popularity is increasing
- Good implementations are within a few percent of the performance of systems with explicit deallocation
- Programmers who like garbage collection feel that the development and debugging time it saves is worth the runtime it costs

Conclusion

- Memory management is an important hidden player in language systems
- Performance and reliability are critical
- Different techniques are difficult to compare, since every run of every program makes different memory demands
- An active area of language systems research and experimentation