CS 322: Languages and Compiler Design II

Mark P Jones, Portland State University

Spring 2015

Week 8: Dynamic memory allocation and garbage collection

Dynamic memory allocation

- Dynamic memory allocation is used when the amount of memory that a program will need at run time cannot easily be predicted when the program is written
- Examples of programs where this is useful include: compilers, web browsers, word processors, ... and many more!

Allocation in run time systems

- Some languages do not support dynamically allocated memory; instead, they require programmers to anticipate/guess memory requirements when they write their programs so that they can pre-allocate enough storage accordingly
- Many operating systems do not support (fine-grained) dynamic memory allocation well ... but many languages require it
- As a result, dynamic memory allocation is one of the most commonly supported features in modern run time systems

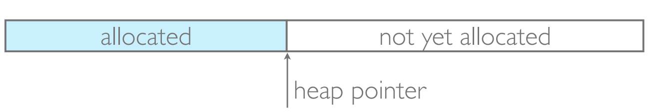
Explicit allocation

Different languages provide different ways to allocate memory:

```
(int*) malloc(120);
new int[30];
new IntExpr(120);
\x -> x+y
(cons x xs)
```


Allocating from a heap

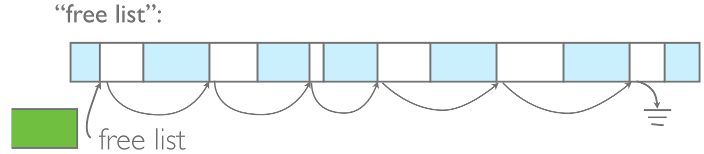
- Where does dynamically allocated memory come from?
- When the run time system is initialized, it requests a large block of memory from the operating system, which is typically referred to as "the heap"
- The run time system maintains a heap pointer that identifies the next free location



- To allocate `n` bytes, we return the current heap pointer value and advance the heap pointer by `n`

Allocating from a free list

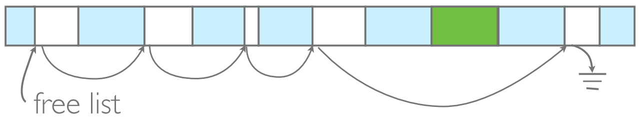
- Unallocated memory areas can be chained together to form a "free list":



- Each block of memory stores (at least) two fields:
  - The length of the block
  - A pointer to the next available block
- To allocate memory, search the free list for the first block that is big enough to hold the corresponding number of bytes

Allocating from a free list

- Unallocated memory areas can be chained together to form a “free list”:



- Each block of memory stores (at least) two fields:
  - The length of the block
  - A pointer to the next available block
- To allocate memory, search the free list for the first block that is big enough to hold the corresponding number of bytes

Allocation is only half the story

- What happens when we run out of memory?
- Can we reclaim and recycle memory when we finish using it?

Reclaiming memory explicitly

- In some languages, programmers can tell the run time system that they have finished with a piece of memory, and that it can be recycled

```
free(bytes);        /* C   */
delete ints;        /* C++ */
```

- This can be risky; the programmer must ensure that:
  - The specified memory was allocated dynamically
  - No part of the program will attempt to access that section of memory again
  - Memory is reclaimed promptly once it becomes unused

Reclaiming memory automatically

- In general, it is hard to know when memory can be reclaimed
  - if memory is reclaimed too early, then the run time system’s structures might be corrupted and the program could crash
  - if memory is reclaimed too late, then the program will have a <u>space leak</u> and use more memory than it needs
- Incorrect attempts to reclaim memory are one of the biggest sources of bugs in C/C++ programs
- Could a run time system do better in deciding when memory can be reclaimed?

Garbage collection

- <u>Garbage collection</u> is the term used to describe automatic reclamation of computer storage
- A memory object is <u>garbage</u> if it will not be used again. In other words, if it is not “<u>live</u>”
- Conceptually, garbage collection is a two phase process
  - <u>Garbage detection</u>: distinguish live memory from memory that is garbage
  - <u>Garbage reclamation</u>: reclaim memory that is garbage so that the running program can reuse it
- In practice, these phases may be interleaved

How do we detect garbage?

- In general, figuring out which sections of memory are garbage is undecidable

is **x** garbage here? we cannot tell!

```
x = new BinExpr(...);
if (... some expression ...) {
    ... something involving x ...
}
return 42;
```

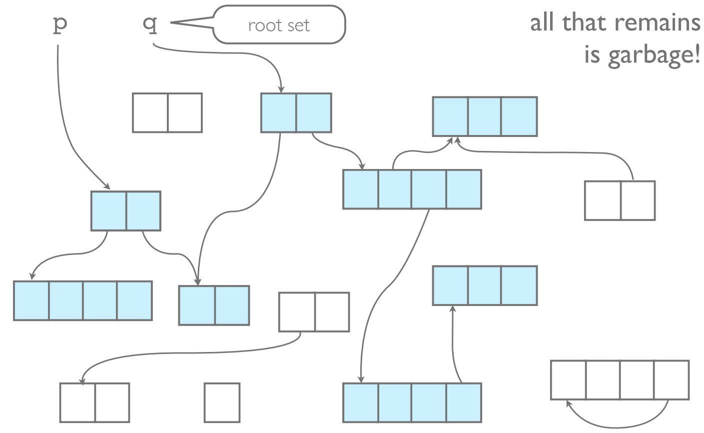
but it is probably garbage here ... unless the code above saved a pointer to it ...

- We will need to approximate!

Reachability

- Suppose that we could interrupt a computation that uses dynamic memory allocation at any point during its execution
- Which objects might be live at that point?
- We can identify a set of roots for live data:
  - Any object that is pointed to by a global variable
  - Any object that is pointed to from an active frame on the stack
- Any object that can be reached from one (or more) of the roots might be used in a future computation
- Objects that cannot be reached are garbage

Understanding reachability



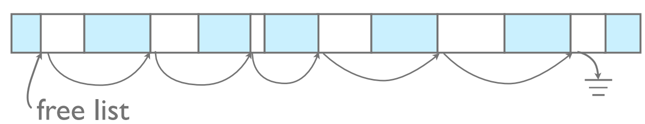
Mark-sweep garbage collection

- This is almost exactly how a mark-sweep garbage collector works:
  - The mark phase: traverse the graph, starting at the roots, and mark every object that is reached
  - The sweep phase: storage for any object that has not been marked can be reclaimed
- The time to garbage collect using this scheme is proportional to the size of the heap: we have to sweep the whole heap to find unmarked objects

How do we reclaim memory?

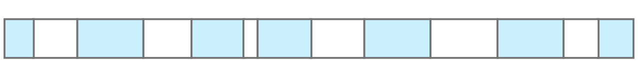
- Once the marking phase is over, the heap will typically be broken in to a mixture of marked and unmarked areas:

- We can reclaim memory by linking together the unused areas:



Fragmentation

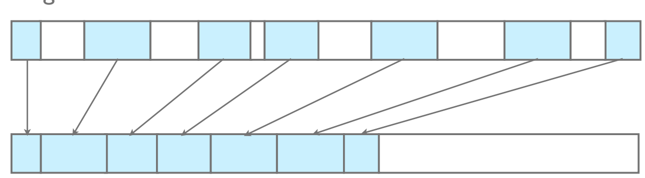
- Another serious problem here is the risk of fragmentation, which happens when memory is broken in to many small pieces that are hard to reuse
- For example, we can't allocate an object  in a heap that looks like this:



- Although there is enough unused memory in total, it isn't available in one contiguous block
- Several compaction techniques have been developed to overcome this problem.

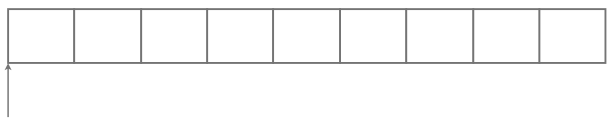
A copying collector

- A copying collector works by copying all of the reachable data in to a safe place, and then discarding the original heap altogether!



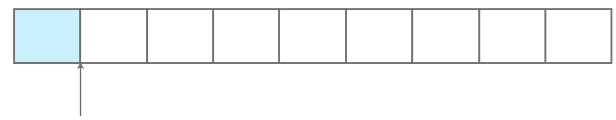
- Copying collectors usually alternate between two heaps
- At each garbage collection, reachable values in one heap (the "from space") are copied to new locations in the new heap (the "to space")

A copying collector walk through



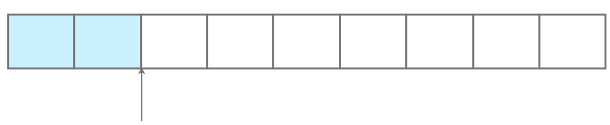
Allocate from a heap ...

A copying collector walk through



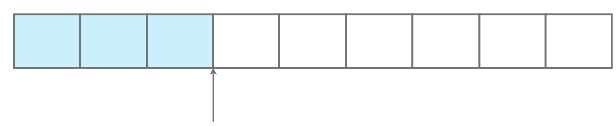
Allocate from a heap ...

A copying collector walk through



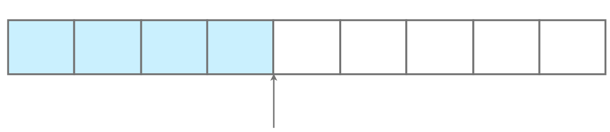
Allocate from a heap ...

A copying collector walk through



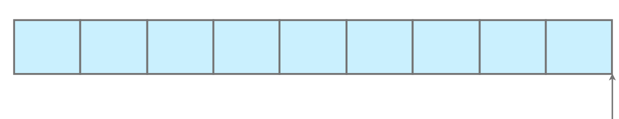
Allocate from a heap ...

A copying collector walk through



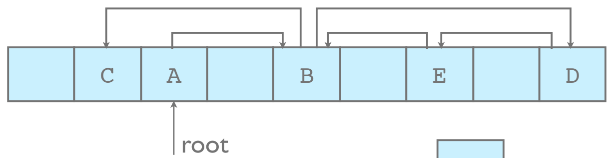
Allocate from a heap ...

A copying collector walk through

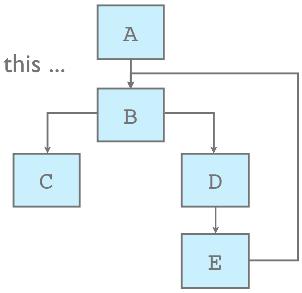


Eventually, the heap is full!

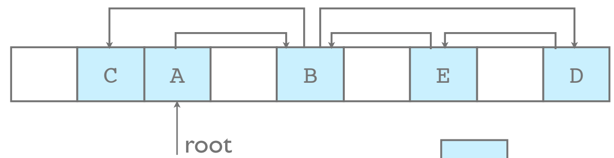
A copying collector walk through



But we might only be using part of this ...

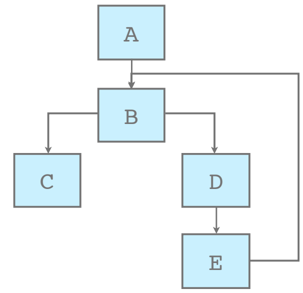


A copying collector walk through

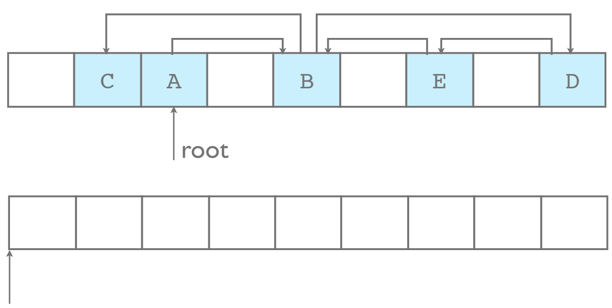


Everything else is "garbage"

Let's stop and copy the live data to a new heap ...



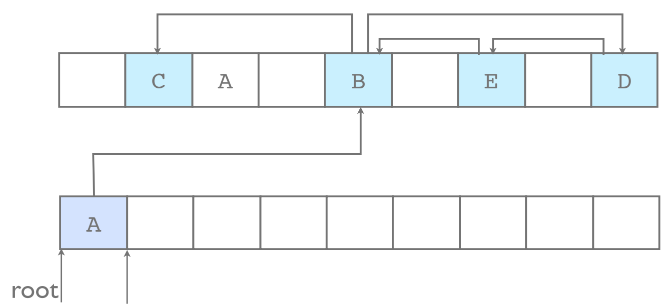
A copying collector walk through



Assume that we have a second block of memory that we can use as a new heap

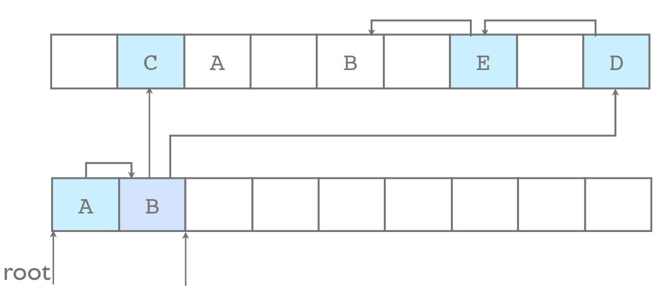
(Algorithm due to Cheney, 1970)

A copying collector walk through



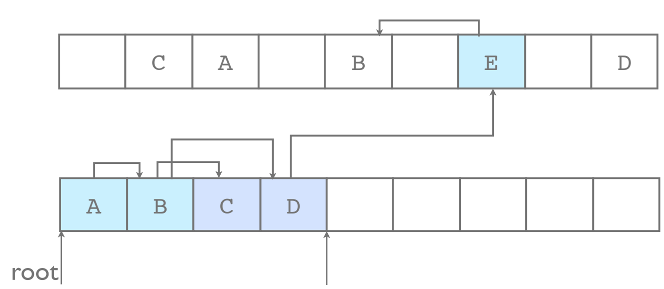
Copy A into the new heap

A copying collector walk through



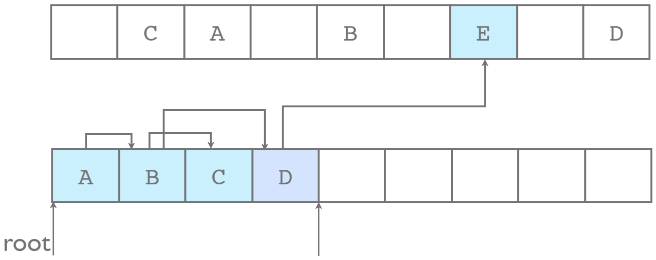
Scavenge A (copy B into the heap)

A copying collector walk through



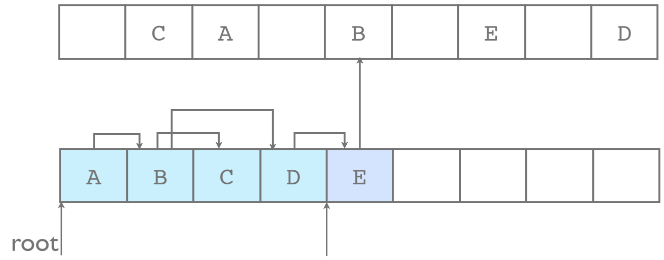
Scavenge B (copy C and D into the heap)

A copying collector walk through



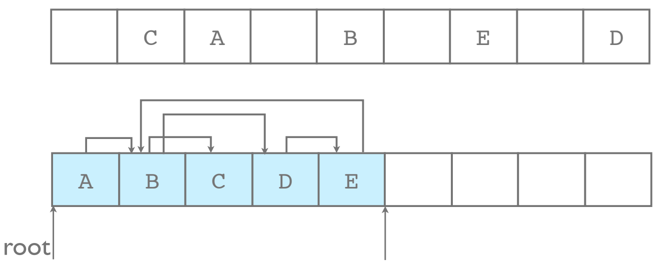
Scavenge C (no objects copied)

A copying collector walk through



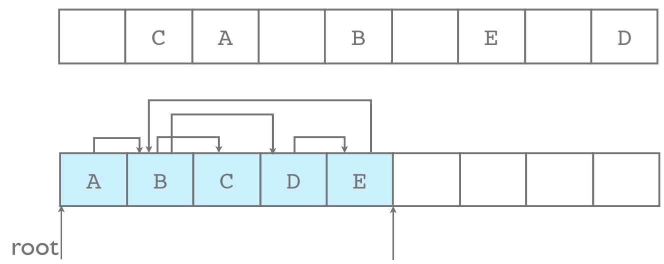
Scavenge D (copy E into the heap)

A copying collector walk through



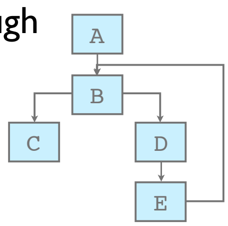
Scavenge E (B is already in the new heap)

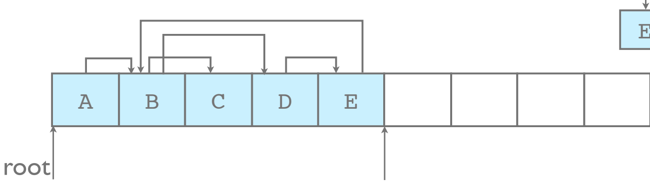
A copying collector walk through



- All live data has been copied to the new heap

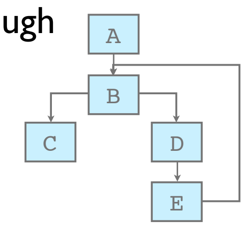
A copying collector walk through

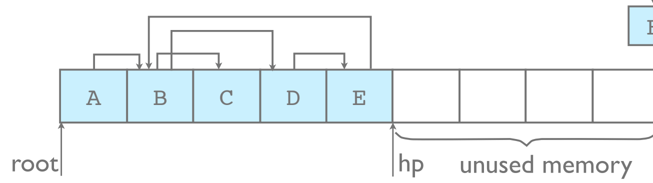




- All live data has been copied to the new heap
- The original data structure has been preserved

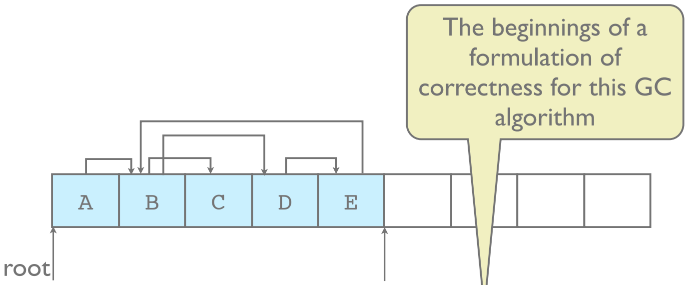
A copying collector walk through





- All live data has been copied to the new heap
- The original data structure has been preserved
- Unused memory reorganized in a single block

A copying collector walk through



- All live data has been copied to the new heap
- The original data structure has been preserved
- Unused memory reorganized in a single block

Pros and cons

✓ **Pro:** A copying collector ensures that the heap is compacted at each garbage collection

- No fragmentation
- We can go back to allocating using a simple heap pointer

✓ **Pro:** The time to garbage collect is proportional to the amount of memory that is reachable, which may be much less than the size of the heap

- **Con:** We have to split available memory resources between two large heaps of equal size, even though we only use one at a time

Some implementation details

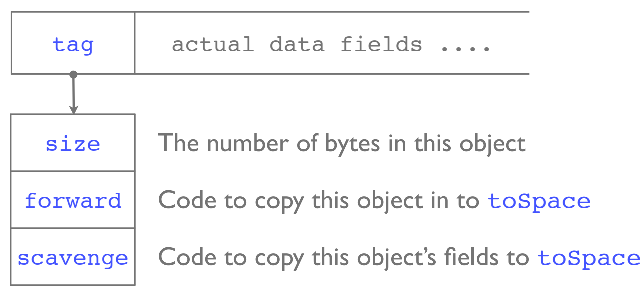
- We will look at a copying garbage collection algorithm in a little more detail
- Let's assume that the runtime system maintains the following variables:

`fromSpace`	the address of the active heap
`toSpace`	the address of the second heap
`hp`	the heap pointer

- Normally, `hp` points into `fromSpace`
- At the start of garbage collection, we reset `hp` to point to the start of `toSpace`

Object representations

- We will also assume that every object that is stored in the heap begins with a pointer to some extra information that is needed to support garbage collection



Forwarding an object

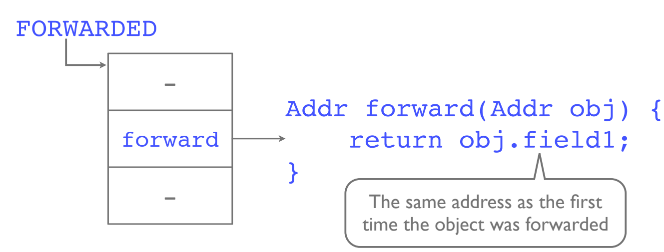
To copy an object from `fromSpace` to `toSpace`:

```
Addr forward(Addr obj) {
    Addr dest = hp;
    for (i=0; i<obj.size; i++) {
        mem[hp++] = obj[i];          Copies fields
    }
    obj.tag    = FORWARDED;
    obj.field1 = dest;               Assumes every object has at least one field
    return dest;
}
```

The object's new address in the `toSpace`

The `FORWARDED` tag

- Once an object has been forwarded, it should not be forwarded again
- We deal with this by overwriting the tag of each forwarded object with the address `FORWARDED`, which points to a special "info table"



Scavenging an object

To forward the fields of an object:

```
void scavenge(Addr obj) {
    obj.field1 = obj.field1.forward();
    obj.field2 = obj.field2.forward();
    ...
}
```

Call the forward() method for this object

Only pointer to objects should be scavenged, and not all fields contain such pointers

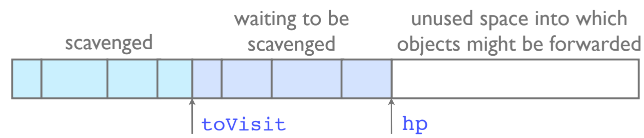
A compiler can generate an appropriate `scavenge()` function for each different type of object that is used in a program using the types of its fields as a guide

Using `toSpace` as a queue

- Initially, `toSpace` is empty:

- Once the roots have been forwarded, it looks like this:

- Now we have scavenge each object, left to right, using `toSpace` as a queue:



Putting it all together

```
hp = toSpace;
for each root r {
    r = r.forward();
}

toVisit = toSpace;
while (toVisit < hp) {
    toVisit.scavenge();
    toVisit += toVisit.size();
}

exchange toSpace and fromSpace;
```

make sure all the roots are forwarded

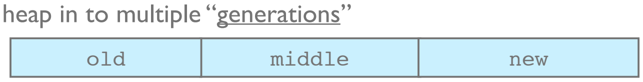
scavenge each forwarded object for pointers

Incremental garbage collection

- The techniques that we have looked at so far put the main computation on "hold" while garbage collection is taking place ("stop and copy")
- For an interactive program with a large heap, this might cause a noticeable pause in execution
- For real-time applications, a long pause is not acceptable
- Much effort has been invested in the design of more sophisticated, "incremental" garbage collection algorithms that solve these problems by interleaving garbage collection with memory allocation

Generational garbage collection

- Experiments suggest most heap-allocated data is short lived
- Generational garbage collectors exploit this by breaking the heap in to multiple "generations"



- The new generation is smaller, takes less time to garbage collect
- Most new objects "die" during the new collection, but those that survive are promoted to the middle generation, which needs less frequent collections
- Objects that survive a middle collection are promoted to the old generation, which needs even less frequent collection

The cost of garbage collection

- Appel has argued that garbage collection can sometimes be cheaper than stack allocation
- Other estimates suggest that use of garbage collection can increase execution time by 10%
- In any case:
  - The cost depends on the quality of the garbage collector, and on the program that uses it
  - There are also overheads with schemes for explicitly reclaimed memory
  - Perhaps the overheads of garbage collection, if there are any, are justified by the resulting reduction in bugs?

Further reading

- There is a large literature on garbage collection; we have only scraped the surface here!
- There is some material on this in Appel's book
- See also "The Garbage Collection Handbook" by Richard Jones (no relation!), Anthony Hosking, and Elliot Moss (details at http://gchandbook.org)
- But you could do a lot worse than start with Paul Wilson's long and dated, but still excellent survey that is available from ftp://ftp.cs.utexas.edu/pub/garbage/bigsurv.ps (or Google for "uniprocessor garbage collection techniques")