

# **Compiler Construction**

## **Lent Term 2014**

### **Lecture 8 (of 16)**

**Read related chapter of Appel**

- **The heap and garbage collection**
  - **Reference counting**
  - **Mark and sweep**
  - **Copy collection**
  - **Generational collection**

**Timothy G. Griffin**  
**[tgg22@cam.ac.uk](mailto:tgg22@cam.ac.uk)**  
**Computer Laboratory**  
**University of Cambridge**

# Memory Management

- Modern programming languages allow programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Memory could easily be exhausted without some method of reclaiming and recycling the storage that will no longer be used.
  - Let programmer worry about it (use **malloc** and **free** in C...)
  - Automatic “garbage collection”

# Solutions

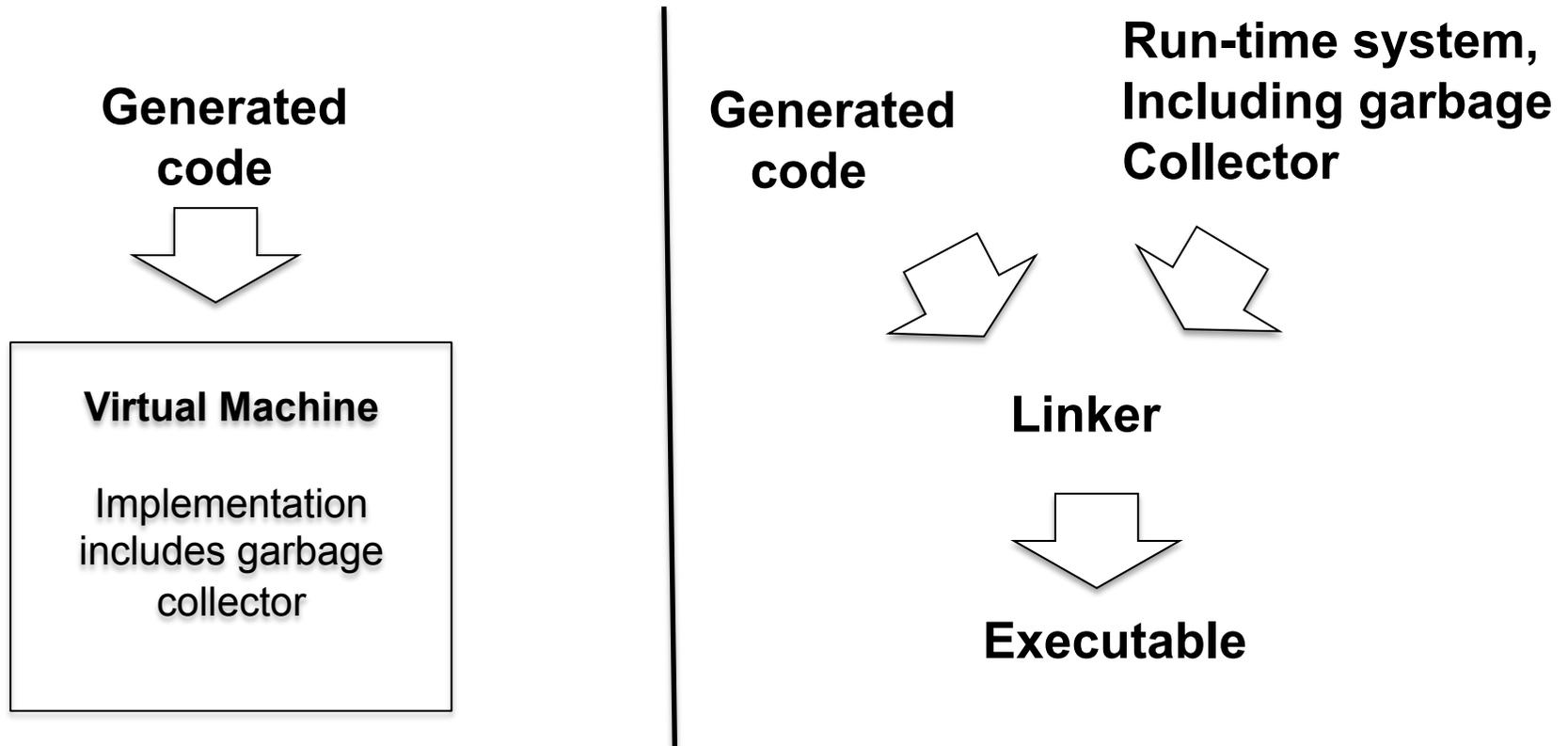
- Let programmer worry about it (use **malloc** and **free** in C...)
- Automatic “garbage collection”
  - Reference Counting
  - Mark and Sweep
  - Copy Collection
  - Generational Collection
  - ... there are other GC techniques...

In general, we must approximate since determining exactly what objects will never be used again is **not decidable**.

# Explicit Memory Management

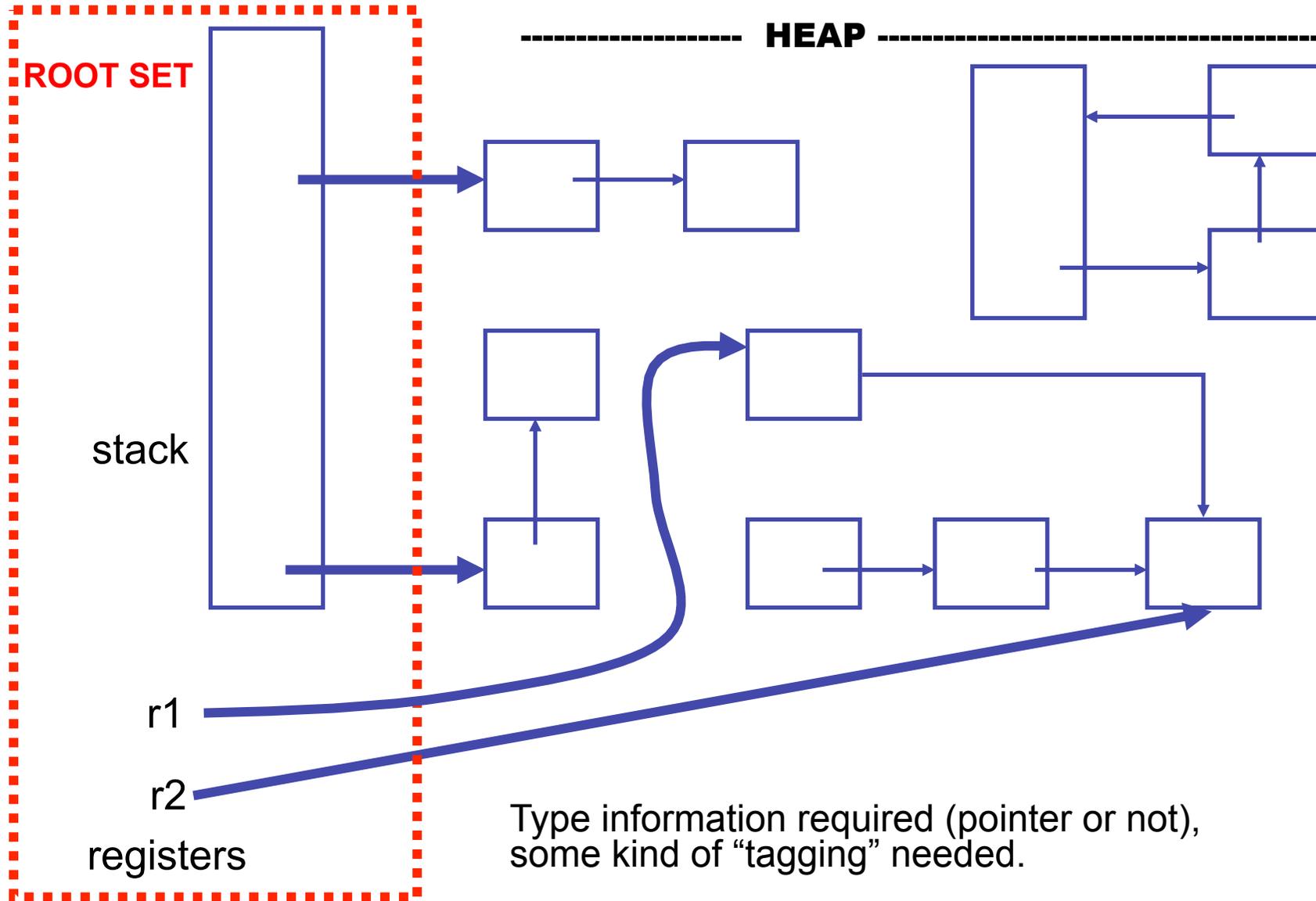
- User library manages memory; programmer decides when and where to allocate and de-allocate
  - `void* malloc(long n)`
  - `void free(void *addr)`
  - Library calls OS for more pages when necessary
  - **Advantage**: Gives programmer a lot of control.
  - **Disadvantage**: people too clever and make mistakes. Getting it right can be costly. And don't we want to automate-away tedium?
  - **Advantage**: With these procedures we can implement garbage collection for “higher level” languages ;-)

# Automatic Memory Management

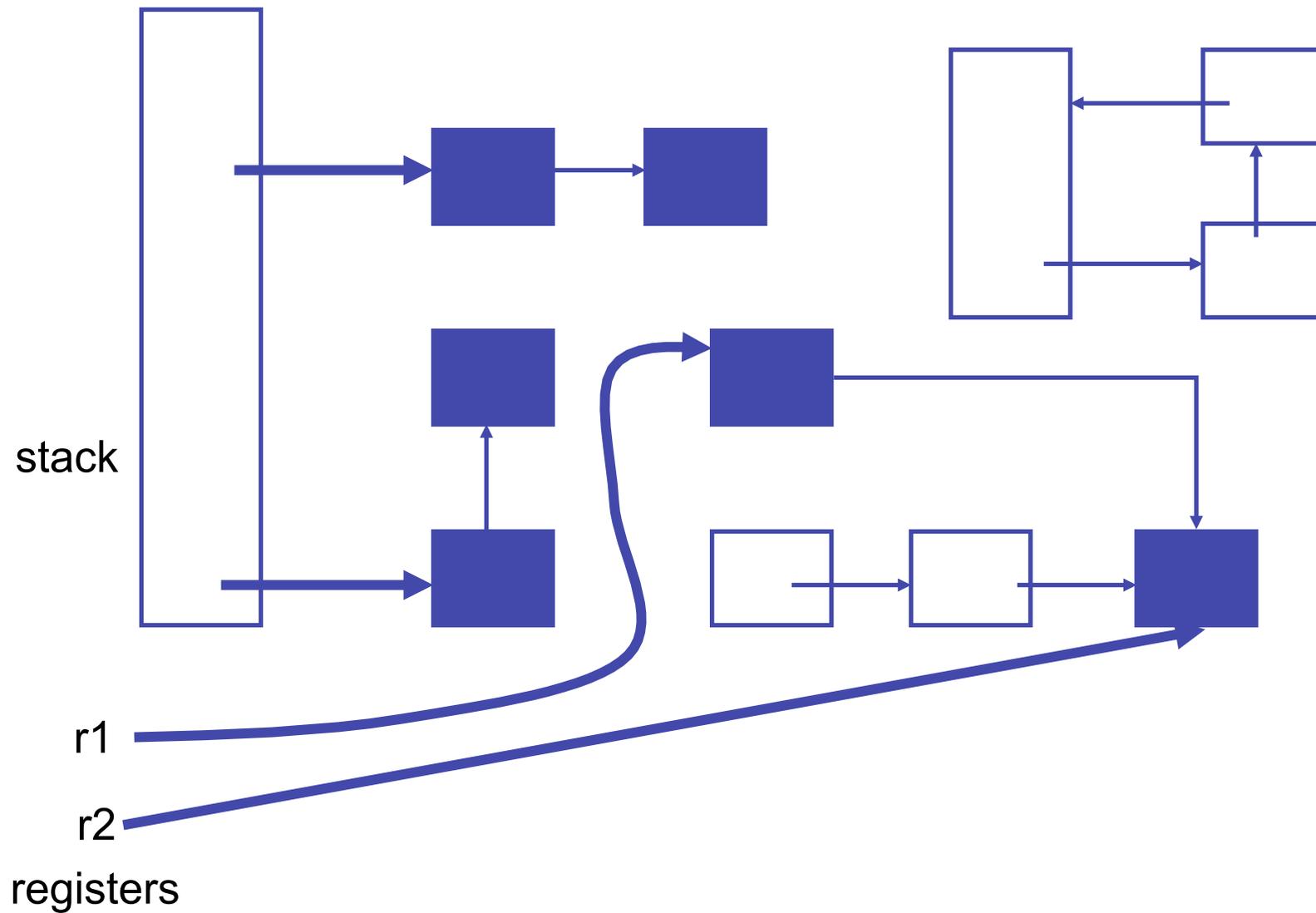


- **When to invoke collection?**
  - **When out of memory?**
  - **When allocating new space?**
  - ...

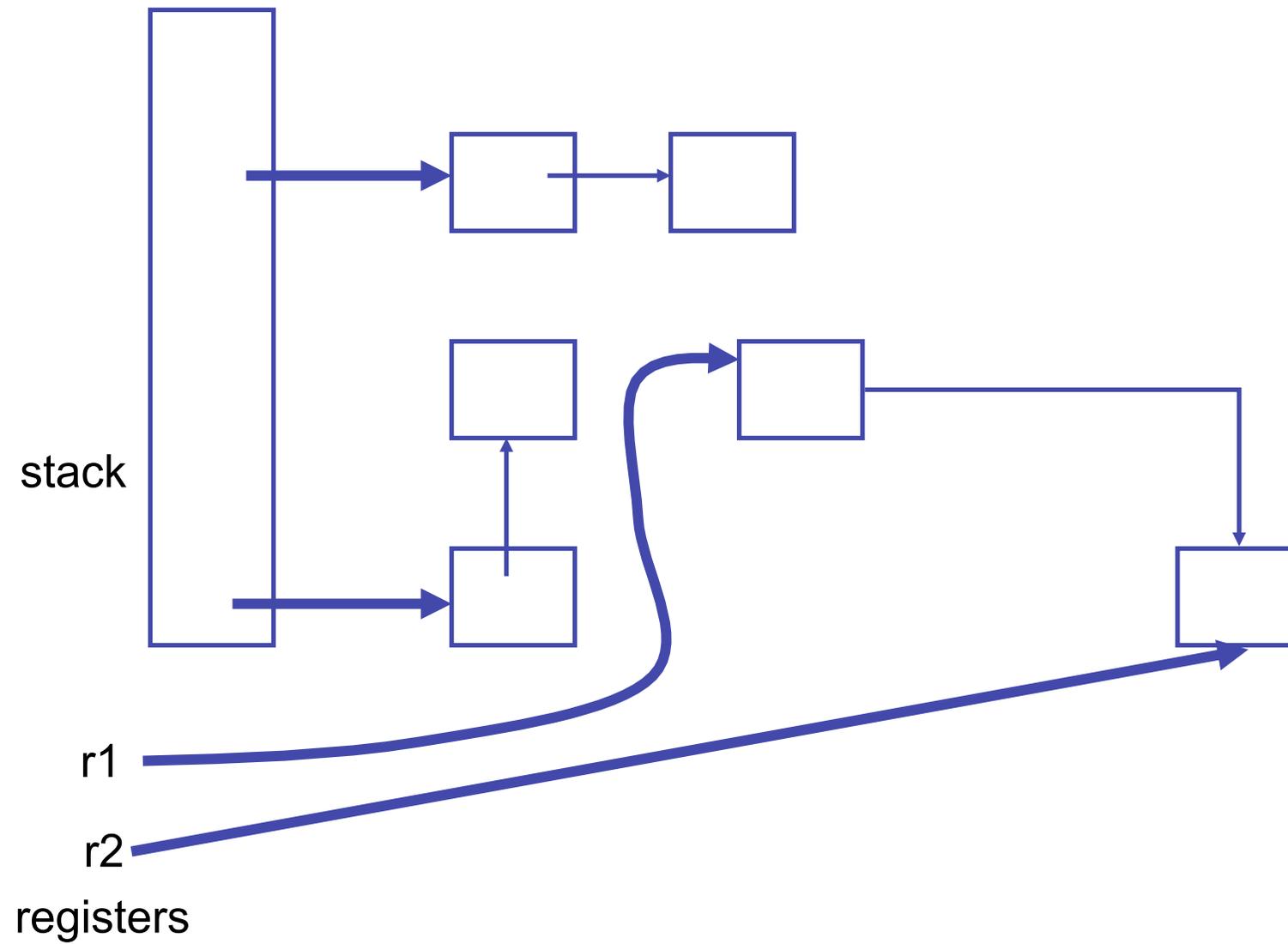
**Automation is based on an approximation : if data can be reached from a root set, then it is not “garbage”**



## ... Identify Cells Reachable From Root Set...



## ... reclaim unreachable cells



## But How? Two basic techniques, and many variations

- **Reference counting** : Keep a reference count with each object that represents the number of pointers to it. Is garbage when count is 0.
- **Tracing** : find all objects reachable from root set. Basically transitive close of pointer graph.

For a very interesting (non-examinable) treatment of this subject see

**A Unified Theory of Garbage Collection.**

David F. Bacon, Perry Cheng, V.T. Rajan.

OOPSLA 2004.

In that paper reference counting and tracing are presented as “dual” approaches, and other techniques are hybrids of the two.

# Reference Counting, basic idea:

- Keep track of the number of pointers to each object (**the reference count**).
- When Object is created, set count to 1.
- Every time a new pointer to the object is created, increment the count.
- Every time an existing pointer to an object is destroyed, decrement the count
- When the reference count goes to 0, the object is unreachable garbage

**Clearly --- this can be VERY costly....**



# Mark and Sweep

- A two-phase algorithm
  - **Mark phase**: Depth first traversal of object graph from the roots to mark live data
  - **Sweep phase**: iterate over entire heap, adding the unmarked data back onto the free list

# Cost of Mark Sweep (somewhat crude)

- Cost of mark phase:
  - $O(R)$  where  $R$  is the # of reachable words
  - Assume cost is  $c1 * R$  ( $c1$  may be 10 instr' s)
- Cost of sweep phase:
  - $O(H)$  where  $H$  is the # of words in entire heap
  - Assume cost is  $c2 * H$  ( $c2$  may be 3 instr' s)
- Analysis
  - The “good” = each collection returns  $H - R$  words reclaimed
  - Amortized cost = time-collecting/amount-reclaimed
    - $((c1 * R) + (c2 * H)) / (H - R)$
    - If  $R$  is close to  $H$ , then each collection reclaims little space..
  - $R / H$  must be sufficiently small or GC cost is high.  
Could dynamically adjust. Say, if  $R / H$  is larger than .5, increase heap size

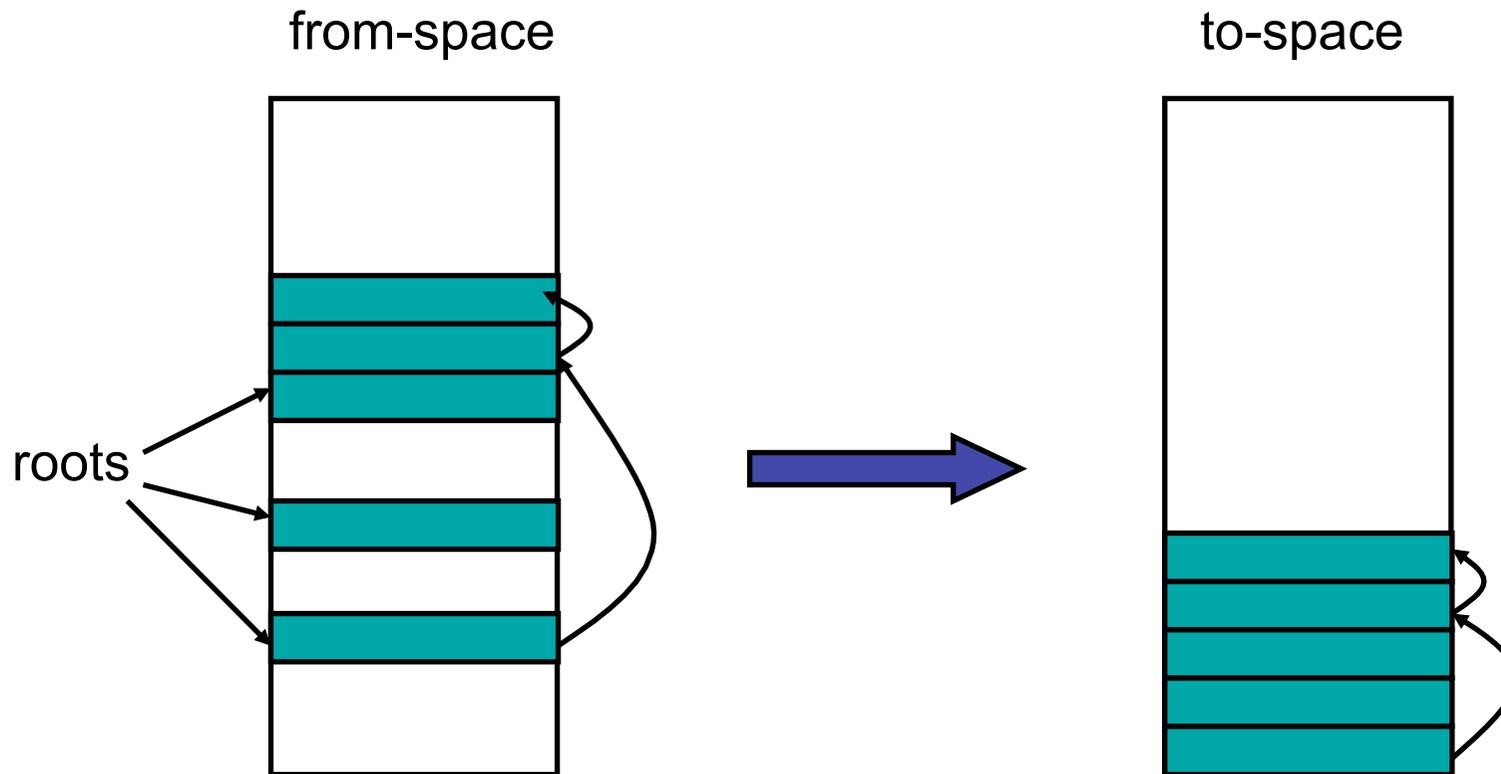
# Other Problems

- Depth-first search is usually implemented as a recursive algorithm
  - Uses stack space proportional to the longest path in the graph of reachable objects
    - one activation record/node in the path
    - activation records are big
  - If the heap is one long linked list, the stack space used in the algorithm will be greater than the heap size!!
  - What do we do? Pointer reversal [See Appel]
- Fragmentation

# Copying Collection

- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

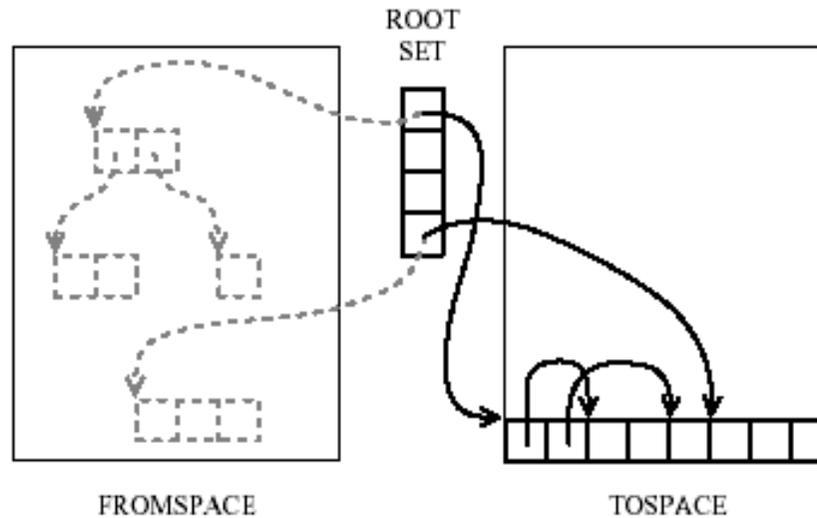
# Copying Collection



# Copying GC

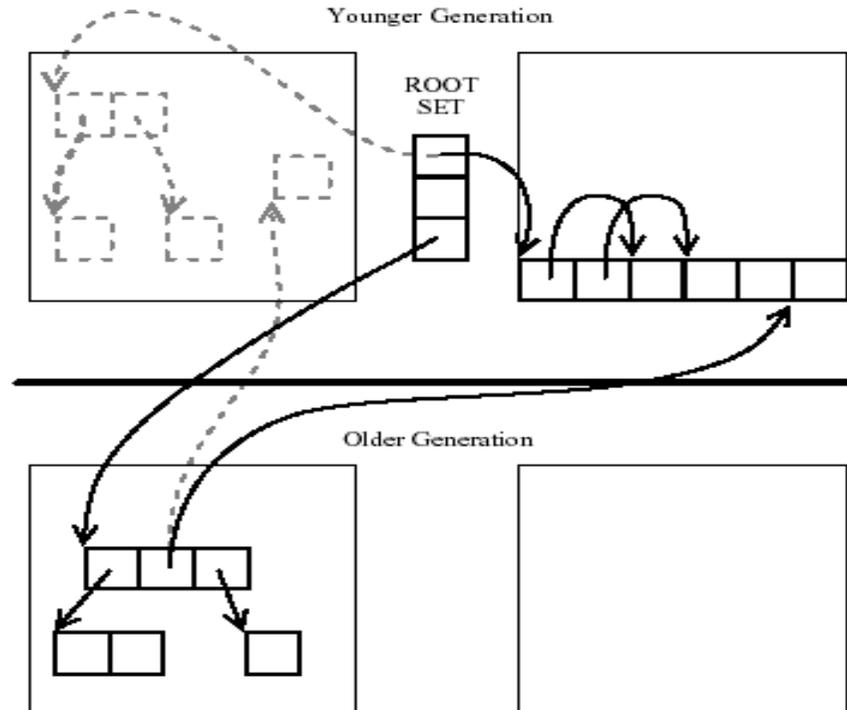
- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
- Cons
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps

## OBSERVATION: for a copying garbage collector



- 80% to 98% new objects die very quickly.
- An object that has survived several collections has a bigger chance to become a long-lived one.
- It's inefficient that long-lived objects be copied over and over.

## IDEA: Generational garbage collection



**Segregate objects into multiple areas by age, and collect areas containing older objects less often than the younger ones.**

# Other issues...

- **When do we **promote** objects from young generation to old generation**
  - **Usually after an object survives a collection, it will be promoted**
- **Need to keep track of older objects pointing to newer ones!**
- **How big should the generations be?**
  - **Appel says each should be exponentially larger than the last**
- **When do we collect the old generation?**
  - **After several **minor collections**, we do a **major collection****
- **Sometimes different GC algorithms are used for the new and older generations.**
  - **Why? Because they have different characteristics**
  - **Copying collection for the new**
    - **Less than 10% of the new data is usually live**
    - **Copying collection cost is proportional to the live data**
  - **Mark-sweep for the old**