

# Algorithms for Dynamic Memory Management (236780) Lecture 1

Lecturer: Erez Petrank

Class on Sundays 10:30-12:30, Taub 4

Reception hours: Thursdays, 14:30

Office 528, phone 829-4942

Web: <http://www.cs.technion.ac.il/~erez/courses/gc>

# Topics today

- Administration
- Overview on memory management:
  - 3 classic algorithms.
- Course topics
- The Mark & Sweep algorithm

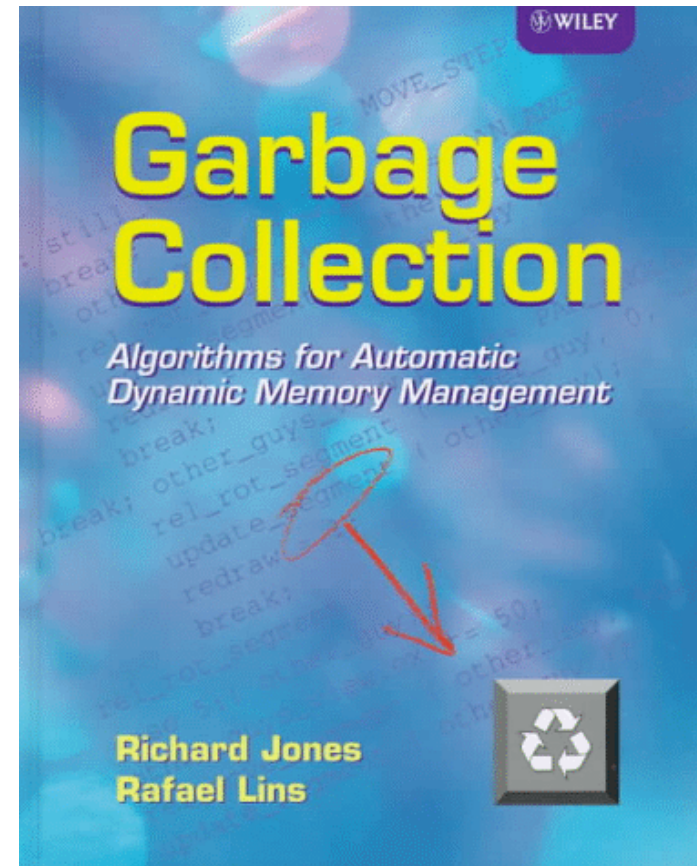
# Grades

- Test (closed material).
- Homework (about 3 (dry) exercises)
- Summary in latex (preferred) or Word
- Grade:
  - Test - at least 75%,
  - Homework - at least 10%
  - Summary - at least 5%
  - Participation in class - at most 10%.

# The book

Garbage Collection:  
Algorithms for  
Automatic Dynamic  
Memory Management  
by Richard Jones,  
Rafael D. Lins

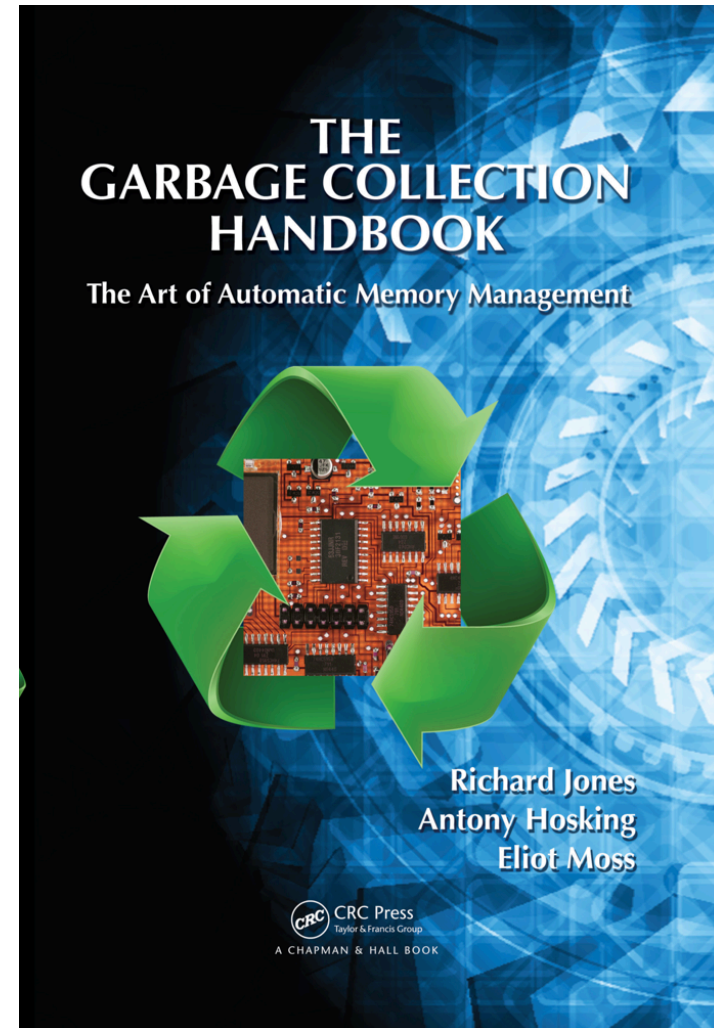
(Available at the library)



# A new book just came out

"The Garbage Collection Handbook: The Art of Automatic Memory Management" by Richard Jones, Anthony Hosking, and Eliot Moss.

(Not yet available at the library)



# The nature of this course

- This is an applied course.
- This is a small elective course.
- Memory management attracts attention from the **programming languages** community, the **algorithms** community, and the **systems** community.
  - We will tend to focus on the algorithmic aspects, system requirements, and parallelism.
  - We'll mention standard engineering techniques.

# The course of this course

- We'll start with simple algorithms.
- Then, their evolution according to studies on "typical" programs/systems behavior.
- Next, we'll cope with the platforms
  - The importance of cache misses
  - Parallel machines
  - Distributed machines
  - Real time requirements
- One theoretical lower bound
- The focus: what is actually used today in the industry.

# Memory Management Overview

- What is it?
- The classical algorithms

# Memory Usage

- All programs use local variables --- static allocation (integer i).
- But sometimes dynamic allocation is required.
  - Usually in more involved programs where length or number of allocated objects depends on the input.
- Memory access is the bottleneck in modern computing systems (both time & energy).

# Dynamic allocation

- All programs use local variables --- static allocation (integer i).
- But sometimes dynamic allocation is required.
  - Usually in more involved programs where length or number of allocated objects depends on the input.
- "Manual" dynamic allocation and de-allocation:

```
Ptr = malloc (256 bytes);  
/* Use ptr */  
Free (Ptr);
```

# Is "manual" management good?

- Practice shows that manual allocation is problematic.
  1. Memory leaks.
  2. Dangling pointers.
- In large projects in which objects are shared by various components of the software, it is sometimes difficult to tell when an object is not needed anymore.

# Solution

- Automatic memory management:
  - User allocates space for an object.

```
course c = new course(236780)
c.class = "TAUB 8"
Faculty.add(c)
```

# Solution

- Automatic memory management:
  - User allocates space for an object.
  - When system "knows" the object will not be used anymore, it reclaims its space.
- "Knows"?
  - Telling whether an object will be accessed after a given line of code is undecidable.
  - A conservative approximation is used instead.

# Solution

- Automatic memory management:
  - User allocates space for an object.
  - When system "knows" the object will not be used anymore, it reclaims its space.
- "Knows"?:
  - Reachability: the program does not have a path of pointers from a program variables to the object.  
(A simple approximation that usually works well.)
  - Other ideas: the compiler can sometimes tell, the user can help with "hints", etc.

# What's good about automatic "garbage collection"?

- Software engineering:
  - Relieves users of the book-keeping burden.
  - Stronger reliability, faster debugging.
  - Code understandable and reliable. (Less interaction between modules.)
- Security (Java):
  - Program never gets a pointer to "play with".

# GC and languages

- Sometimes it's built in:
  - LISP, Java, C#.
  - The user cannot free an object.
- Sometimes it's an added feature:
  - C, C++.
  - User can choose to free objects or not. The collector frees all objects not freed by the user.
- Most modern languages are supported by garbage collection.



# Most modern languages rely on GC

## Well-known languages supported by garbage collection

ActionScript (2000)	Algol-68 (1965)	AppleScript (1993)
AspectJ (2001)	Beta (1983)	C# (1999)
Managed-C++ (2002)	Cecil (1992)	Clean (1984)
CLU (1974)	D (2007)	Dylan (1992)
Dynace (1993)	Eiffel (1986)	Elastic-C (1997)
Emerald (1988)	Erlang (1990)	Euphoria (1993)
F# (2005)	Green (2005?)	Groovy (2004)
Haskell (1990)	Hope (1978)	Icon (1977)
Java (1994)	JavaScript (1994)	Liana (1991)
Limbo (1996)	Lingo (1991)	LotusScript (1995)
Lua (1994)	Mercury (1993)	Modula-3 (1988)
Miranda (1985)	ML (1990)	Oberon (1985)
Objective-C (2007)	Obliq (1993)	Perl (1986)
PHP (1995)	Pliant	PostScript (1982)
Python (1991)	Rexx (1979)	Ruby (1993)
Sather (1990)	Scala (2003)	Scheme (1975)
Self (1986)	SETL (1969)	Simula (1964)
SISAL (1990)	Smalltalk (1972)	Snobol (1962)
Squeak (1996)	tcl (1990)	Theta (1994)
VB.NET (2002)	VBScript (1995?)	VHDL (1987)
	YAFL	

## Languages developed since 1990 not supported by garbage collection

Alef (1995)	AutoIt (1999)	Befunge (1993)
Cilk (1995)	Delphi (partly, 1995)	Goedel (1994)
	Visual Basic (1991)	

# What's bad about automatic "garbage collection"?

- It has a cost:
  - Old Lisp systems 40%.
  - Today's Java program (if the collection is done "right") 5-15%.
- Considered a major factor determining program efficiency.
- Techniques have evolved since the 60's. In this course we investigate the ideas developed from then until now.

# What kind of area is this?

Here algorithms “touch” systems.

- On one hand - must know the system well to design a good algorithm.
- On the other hand - should be able to understand an algorithm to get good system performance.

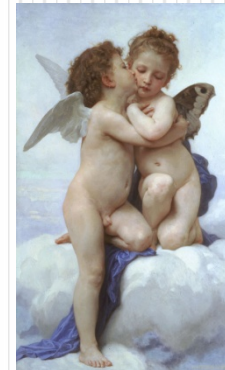
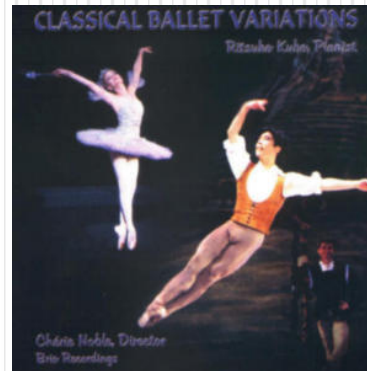
# How have the techniques evolved?

- Hard to compare collection algorithms.
- Asymptotic analysis not relevant.
- Implementation is sometimes more important than the algorithm.
  
- But - good ideas caught.

# Note:

- We will discuss memory management in the context of a programming language (mostly with Java).
- The ideas are useful for
  - other languages
  - operating systems memory management
  - disks management, etc.

# The Classical Algorithms

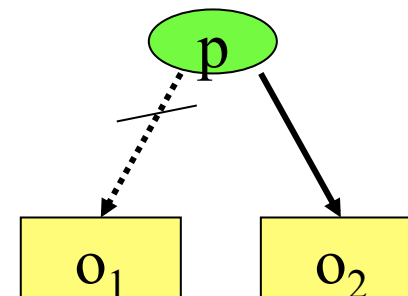


# Reference counting [Collins 1960]

- Recall that we would like to know if an object is reachable from the roots.
- Associate a **reference count** field with each object: how many pointers reference this object.
- When nothing points to an object, it can be deleted.
- Very simple, used in many systems.

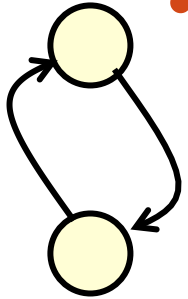
# Basic Reference Counting

- Each object has an RC field, new objects get  $o.RC:=1$ .
- When  $p$  that points to  $o_1$  is modified to point to  $o_2$  we execute:  $o_1.RC--$ ,  $o_2.RC++$ .
- if then  $o_1.RC==0$ :
  - Delete  $o_1$ .
  - Decrement  $o.RC$  for all "children" of  $o_1$ .
  - Recursively delete objects whose RC is decremented to 0.



## 3 years later...

- [Harold-McBeth 1963] The Reference counting algorithm does not reclaim cycles!



- But:
  - "Normal" programs do not use too many cycles.
  - So, other methods are used "infrequently" to collect the cycles.

# During the years

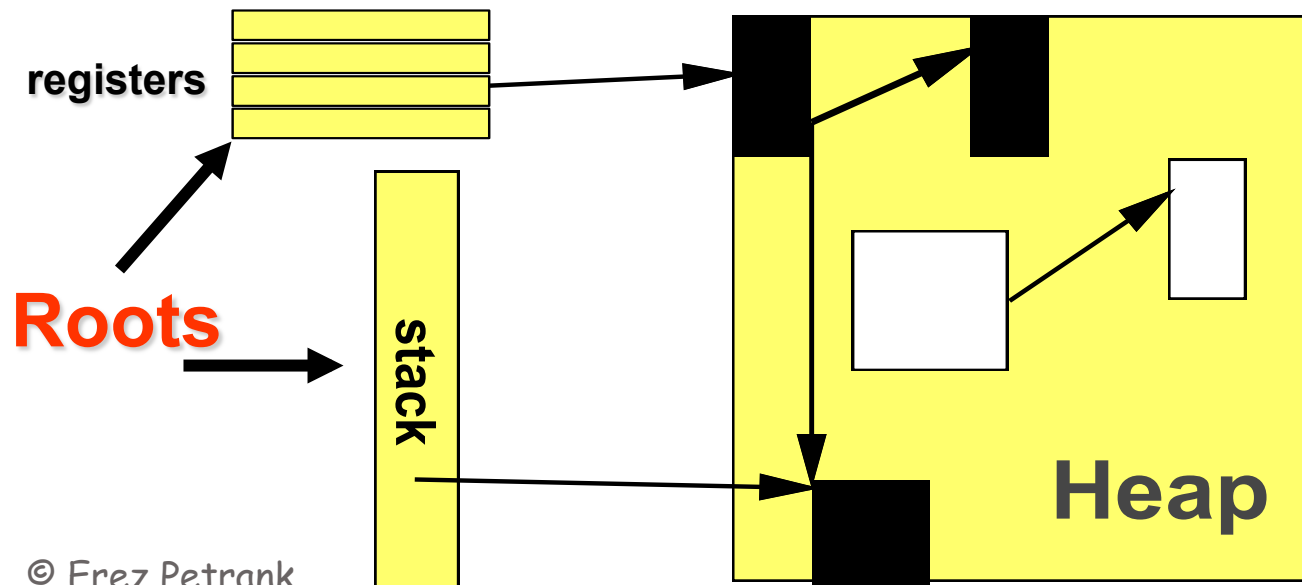
- Many improvements over the basic reference counting method.
- Cycle collection algorithms.
- To be discussed in a separate lecture.

# The Mark-and-Sweep Algorithm [McCarthy 1960]

- Mark phase:
  - Start from **roots** and traverse all objects **reachable** by a path of pointers.
  - Mark all traversed objects.
- Sweep phase:
  - Go over all objects in the heap.
  - Reclaim objects that are not marked.

# The Mark-Sweep algorithm

- Traverse live objects & mark black.
- White objects can be reclaimed.

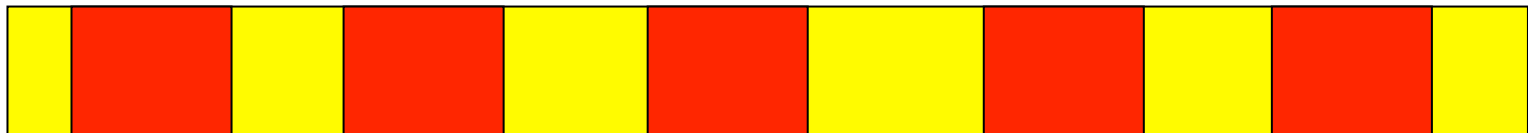


Note!  
This is not  
the **heap**  
data  
structure!

# Mark-Compact

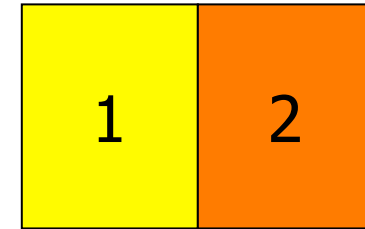
- Gradually, the **heap** gets **fragmented**.
- When space is too fragmented to allocate, a compaction algorithm is used.
- We will see several compactors. Issues:
  - Keep order of objects?
  - Use extra space?
  - How many heap passes?
  - Parallelism?

The  
Heap

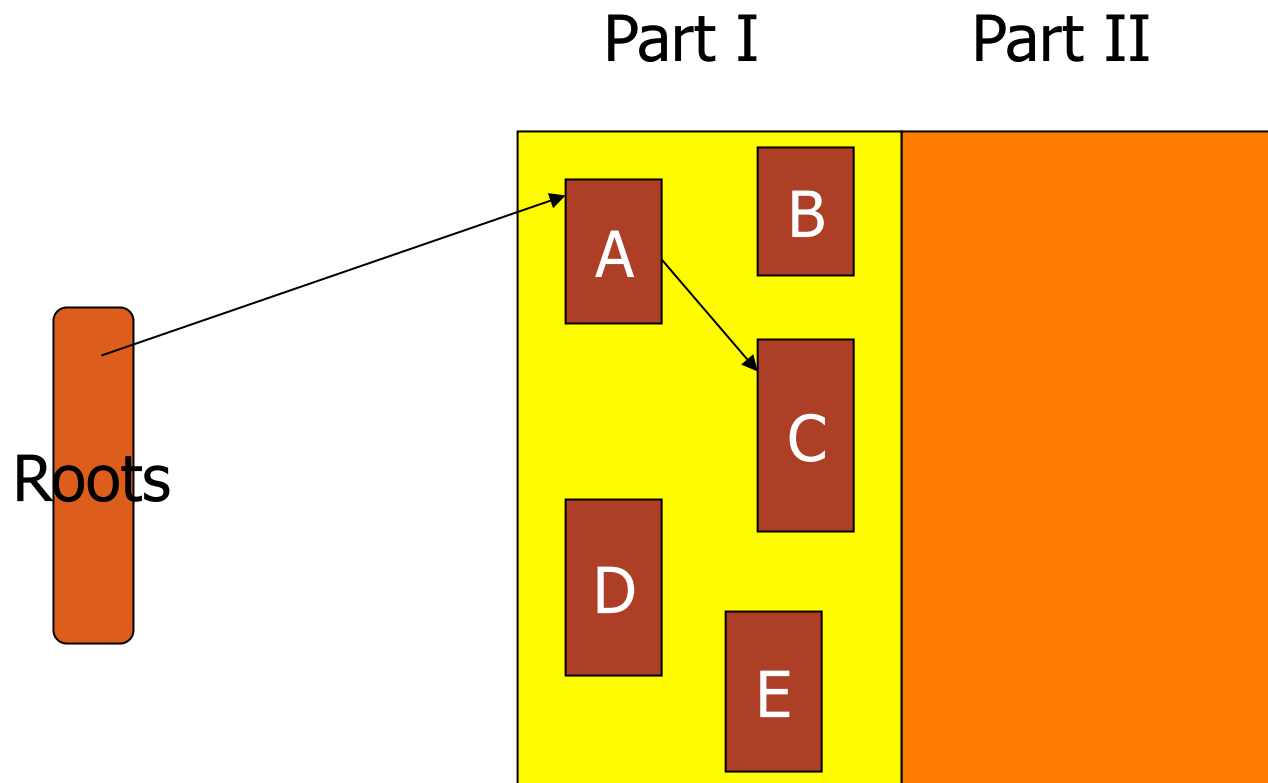


# Copying garbage collection

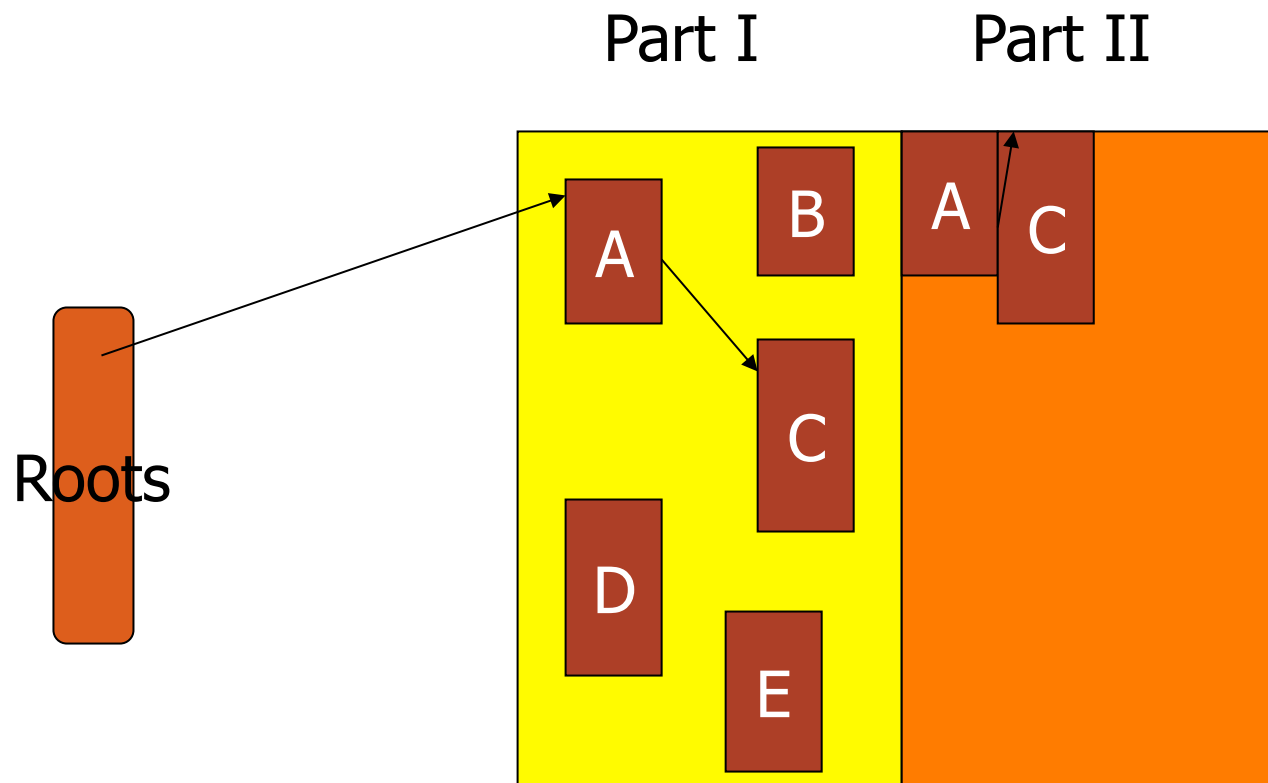
- **Heap** partitioned into two.
- Part 1 takes all allocations.
- Part 2 is reserved.
- During *GC*, the collector traces all reachable objects and copies them to the reserved part.
- After copying, activity goes to part 2. Part 1 is reserved till next collection.



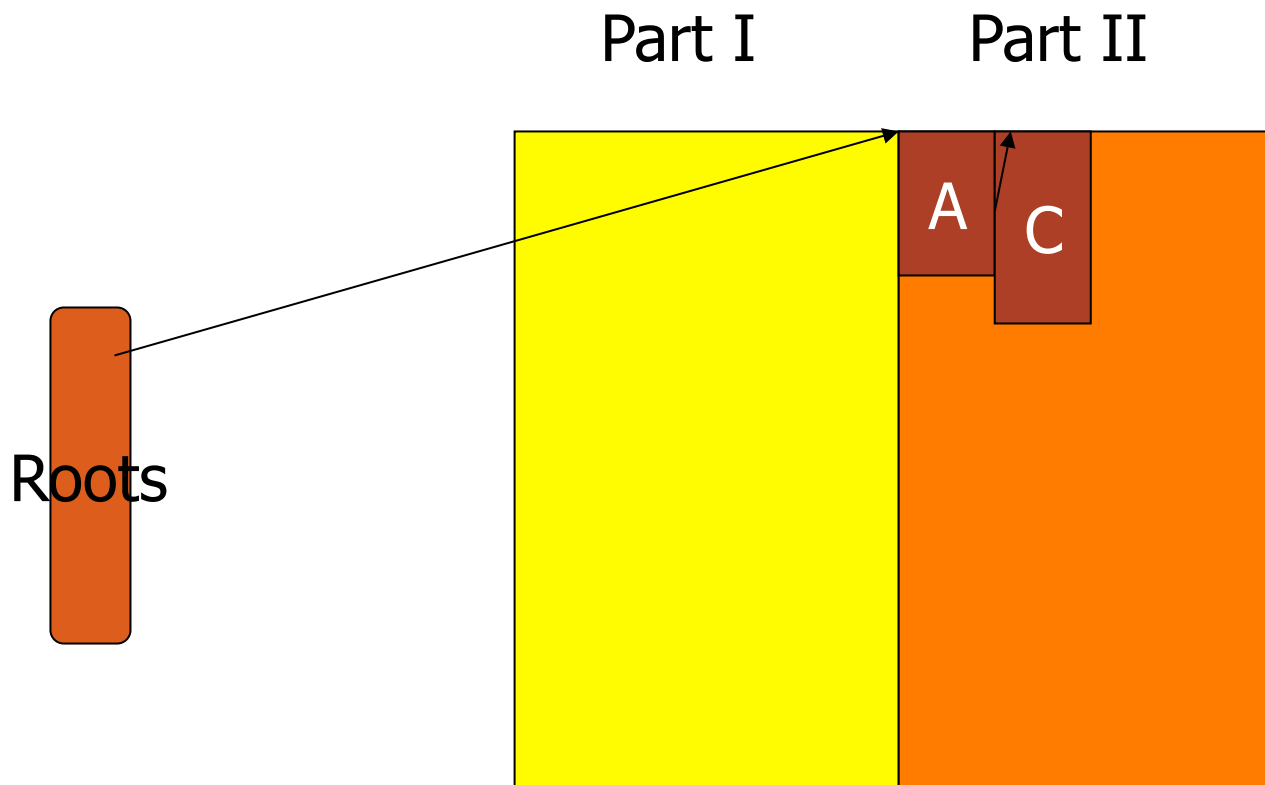
# Copying garbage collection



# The collection copies...



Roots are updated; Part I reclaimed.



# Properties

- Compaction for free
- Major disadvantage: half of the heap is not used.
- "Touch" only the live objects
  - Good when most objects are dead.

## A very simplistic comparison

	Reference Counting	Mark & sweep	Copying
Complexity	Pointer updates + dead objects	Size of heap (live objects)	Live objects
Space overhead	Count/object + stack for DFS	Bit/object + stack for DFS	Half heap wasted
Compaction	Additional work	Additional work	For free
Pause time	Mostly short	long	long
More issues	Cycle collection		

# Some terms to be remembered

- Heap, objects
- Allocate, free (deallocate, delete, reclaim)
- Reachable, live, dead, unreachable
- Roots
- Reference counting, mark and sweep, copying, tracing algorithms
- Fragmentation

# Course Topics



# Course Topics

- Introduction + Mark and Sweep algorithms
- Compaction algorithms
- Copying algorithms
  - Baker's incremental copying [1978].
- Generational collectors
  - The Train algorithm.
- Concurrent & On-the-fly algorithms
  - Mostly concurrent collection (SUN, IBM, BEA, others?) .
  - On the fly collection.
- Parallel collection.

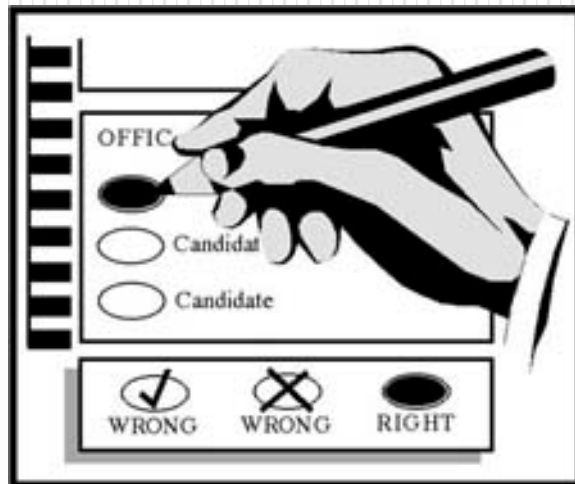
# Course Topics

- Snapshot & Sliding Views collections.
- Reference Counting collectors (also sliding views).
- Cycle Collection.
- Allocation techniques.
- Parallel and concurrent compaction.
- Distributed garbage collection.
- Real-time support.
- Cache-conscious garbage collection:
  - Including impossibility result.

# More Possible Topics

- Conservative garbage collection
- Worst-case fragmentation.
- Verifying memory managers.
- Related compiler and runtime issues
- Age-oriented collection
- Age-based Garbage Collection

# Memory Management: Class II: Mark & Sweep



# The basic idea [McCarthy 1960]:

1. Mark all the reachable objects from the roots.
2. Scan all the heap and reclaim all unmarked objects.

John McCarthy  
(1927--2011)  
in PLDI 2002



# Triggering

```
New(A)=  
  if no available allocation space  
    mark_sweep()  
    if no available allocation space  
      return ("out-of-memory")  
  pointer = allocate(A)  
  return (pointer)
```

# Basic Algorithm (Cont.)

```
mark_sweep()=  
for Ptr in Roots  
    mark(Ptr)  
sweep()
```

```
mark(Obj)=  
if mark_bit(Obj) == unmarked  
    mark_bit(Obj)=marked  
for C in Children(Obj)  
    mark(C)
```

```
Sweep()=  
p = Heap_bottom  
while (p < Heap_top)  
    if (mark_bit(p) == unmarked) then free(p)  
    else mark_bit(p) = unmarked;  
    p=p+size(p)
```

# Properties of Mark & Sweep

- Does not move objects:
  - ☺ *Conservative* collection possible (when pointers cannot be accurately identified).
  - ☹ Fragmentation
- Complexity:
  - ☺ Mark phase: live objects (dominant phase)
  - ☹ Sweep phase: heap size.
- Termination: each pointer traversed once.
- Most popular method today (at a more advanced form).

# Standard Engineering Techniques

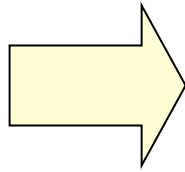
- Mark:
  - Explicit mark-stack (avoid recursion)
  - Pointer reversal
  - Using bitmaps
- Sweep:
  - Using bitmaps
  - Lazy sweep

# Making recursion explicit

- Problem: if object graph is “unsuitable”, then recursion becomes too deep.
  - Large space overhead (compiler stack frames)
  - Inefficient execution (function calls)
  - Potential crash (user does not understand why)
- Solution: use iterative loops and auxiliary data structure instead of functions calls.

# Modifying mark-roots

```
mark_sweep()=  
for ptr in Roots  
    mark(ptr)  
sweep()
```



```
mark_sweep()=  
mark_stack = empty  
for obj referenced by Roots  
    mark_bit(obj) = marked  
    push(obj, mark_stack)  
Mark_heap()  
sweep()
```

# Modifying heap scan

```
mark_heap()=
while mark_stack != empty
  obj = pop (mark_stack)
  for C in children (obj)
    if mark_bit (C) == unmarked
      mark_bit (C) = marked
      push (*C, mark_stack)
```

```
mark(Obj)=
if mark_bit(Obj) == unmarked
  mark_bit(Obj) = marked
  for C in Children(Obj)
    mark(C)
```

# Implementing the stack

- "Standard" implementation: linked list of small `mark_stacks`.
- **Dealing with markstack overflow:**
  - Using cyclic stack (Knuth) or dropping overflowed objects (Bohem *et al*)
  - After dropping objects from the stack, it is enough to go over the heap to find mark -> unmarked references.



# Where is the mark-bit ?

- Option 1: within the object.
  - Problems: locality during sweep, space in object, concurrency of collector-program.
- Option 2: Keep a mark-bit table.
  - Associate the object and the bit by address translation, meaning fixed space per object (most popular).
  - Almost a must for sweep:  
Go over a short table and find sequences of zero's.

# Lazy Sweeping

Cache behavior is a serious issue!

- Sweep requires time in which the program is halted, and even when using a bitmap, creates a **cache-hazard**.
- Can we break sweep into small segments of work?
- Lazy sweep [Hughes]: when allocating, sweep until finding an appropriate free space.
  - Better locality
  - Shorter pauses.

# Bitwise Sweep

- A major overhead of sweep is finding & freeing each chunk of 0's in the mark-bit table.
- **Solution:** free only large chunks of objects.
  - Sweep uses only full zero words (fast search).
  - When found - free the entire chunk of zeros (not only the space presented by the zero word).
- Advantage: looking for a zero word is fast.
- Disadvantage: loses some space (but only small chunks).

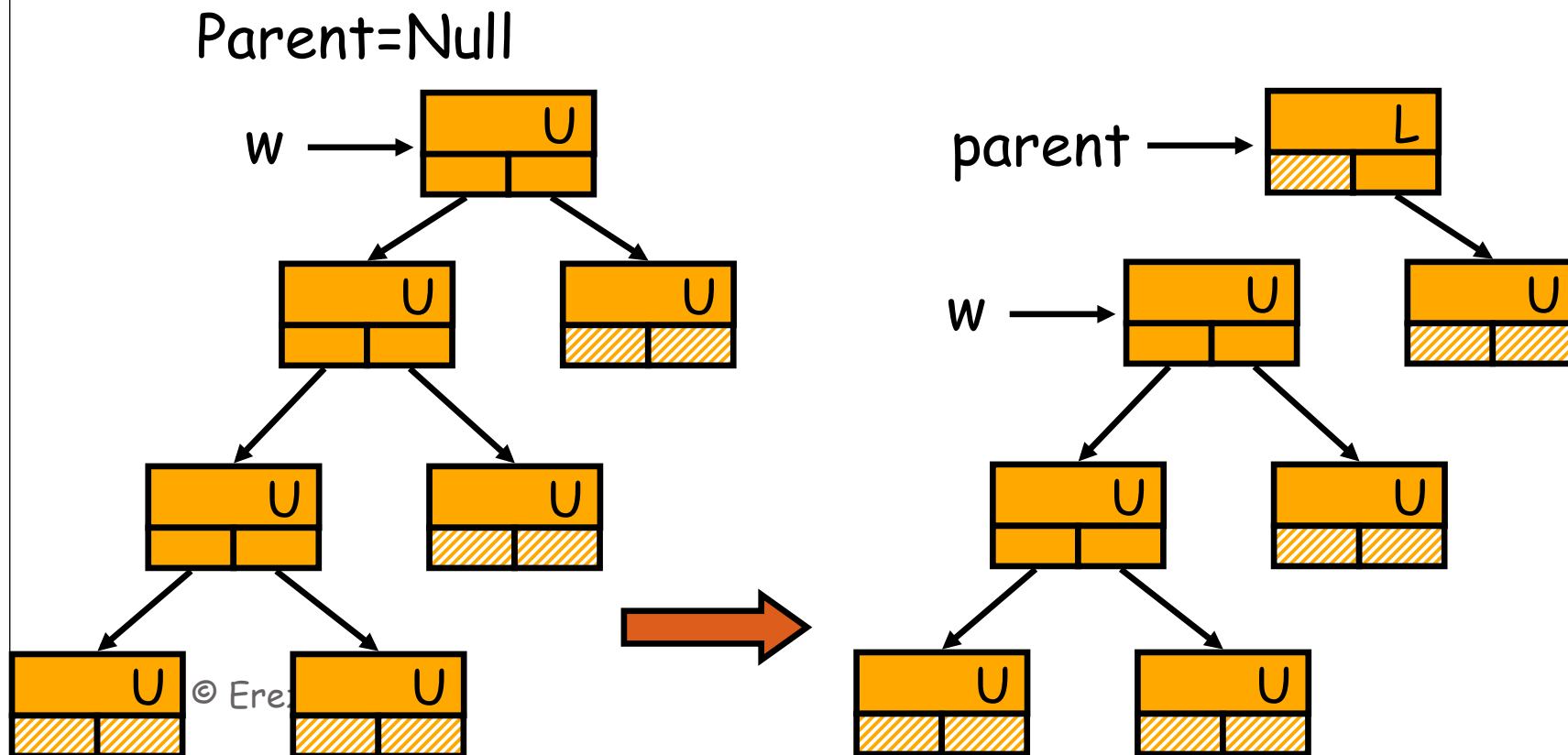
# Pointer Reversal

(The following is an excerpt from the Data Structures course)

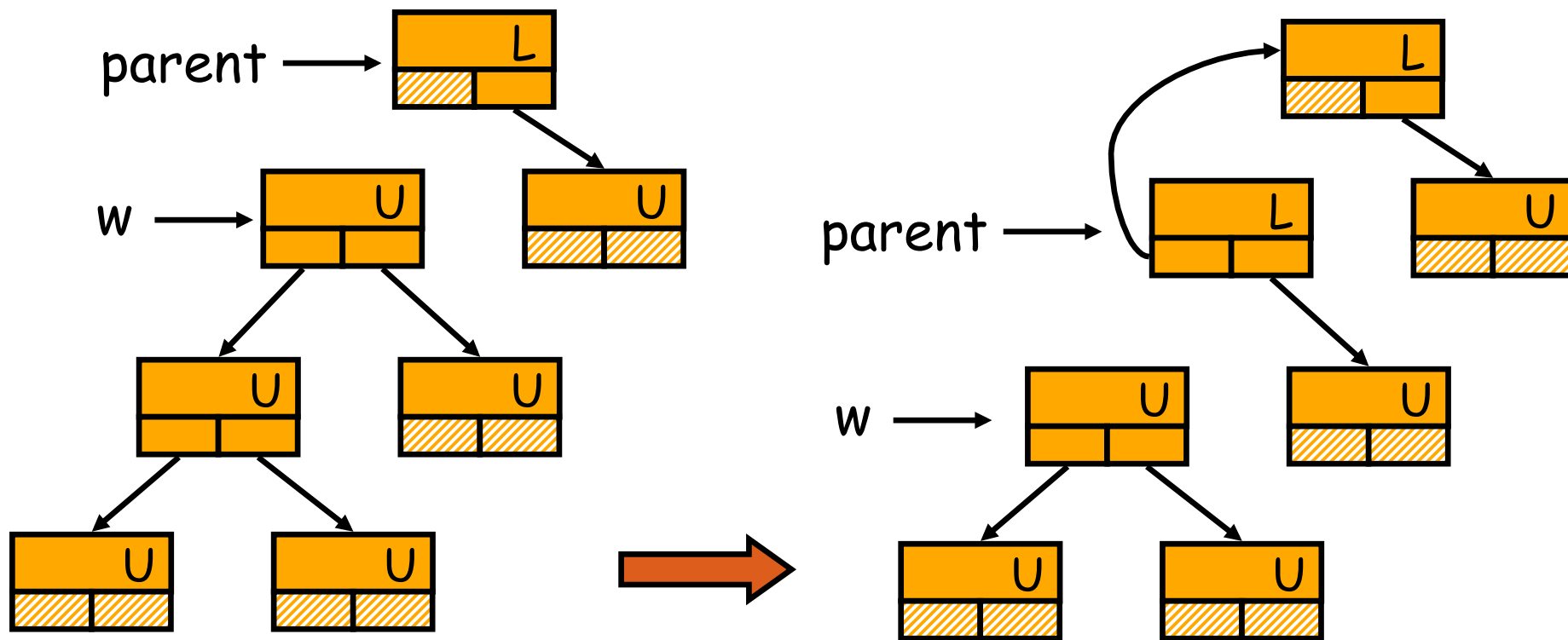
- Marking with “no extra auxiliary space”.
- Mark info kept inside objects.
- Stack replaced by reversing pointers during graph traversal.

# סימון ללא רקורסיה

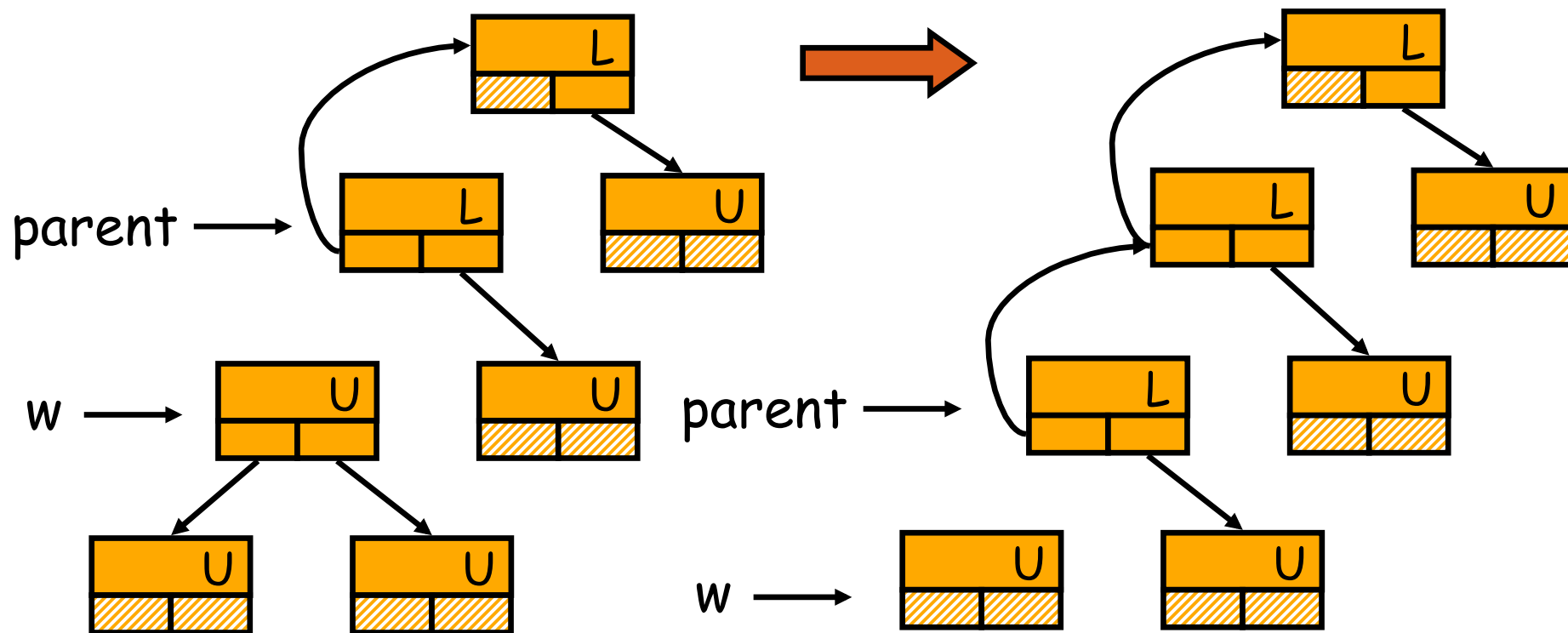
הרעיון: כאשר נבקר בצומת, נשמור את המכוון להורה שלו בשדה left או right כך שנוכל לחזור להורה בסוף הביקור. לתווית של כל צומת יהיו שלושה ערכים:  $\{l, r, u\}$  שיאפשרו לדעת איזה מכוון מצביע להורה. נשתמש במשתנים הבאים:  $root$  עבור שורש העץ ו- $w$  עבור הצומת הנוכחי (מאותחל ל- $root$ ). השטה נקראת הפוך מצביעים (Pointer Reversal).



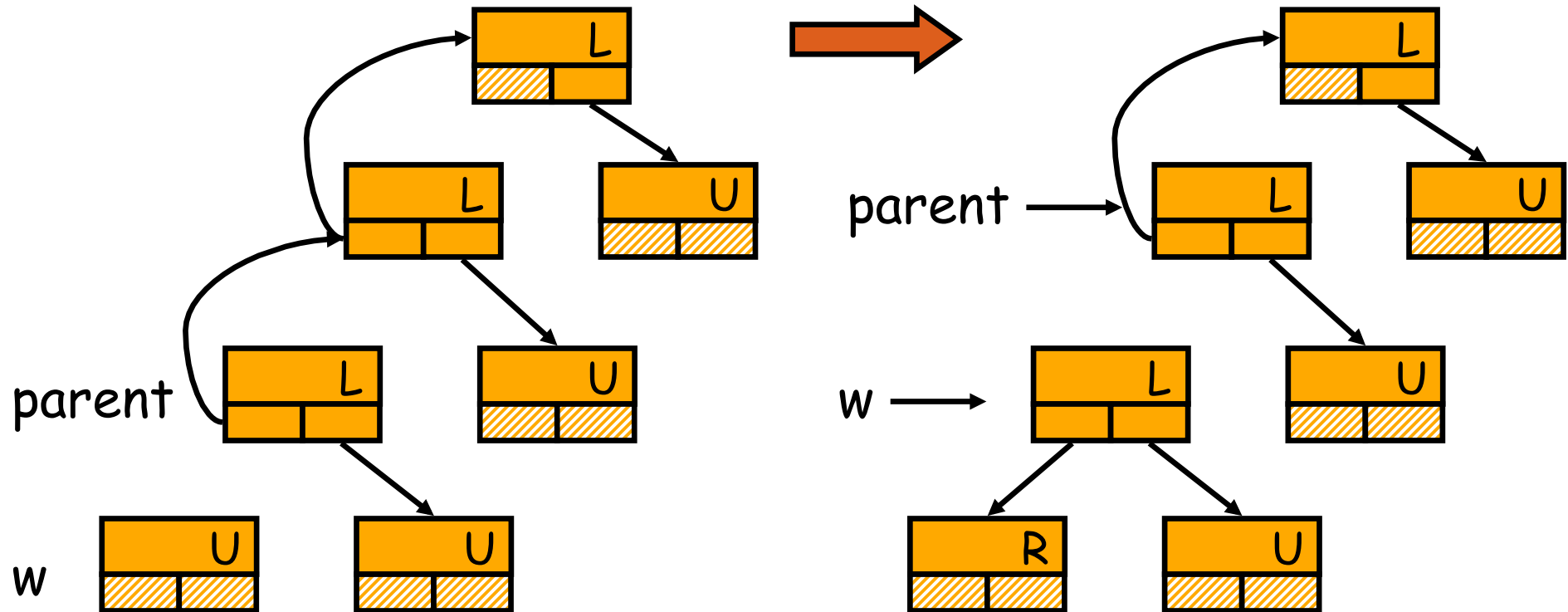
# סימון ללא רקורסיה (המשך)



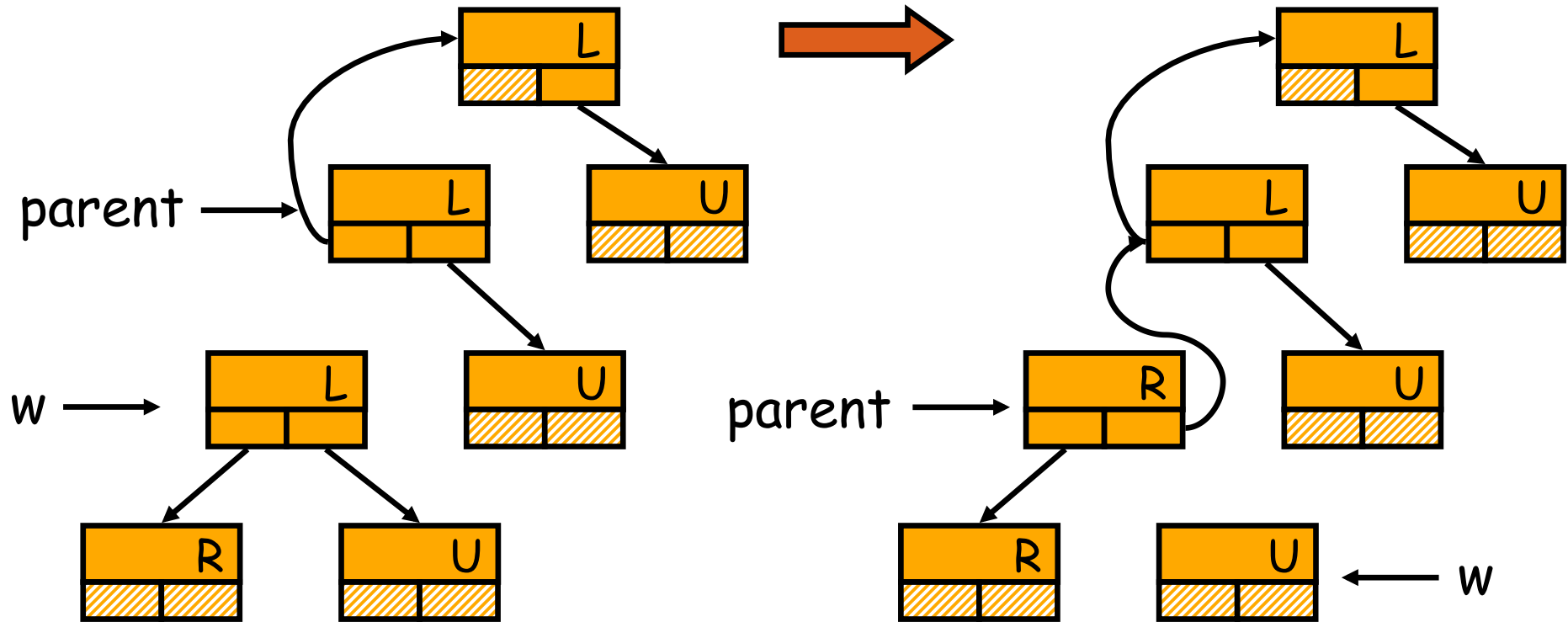
# סימון ללא רקורסיה (המשך)



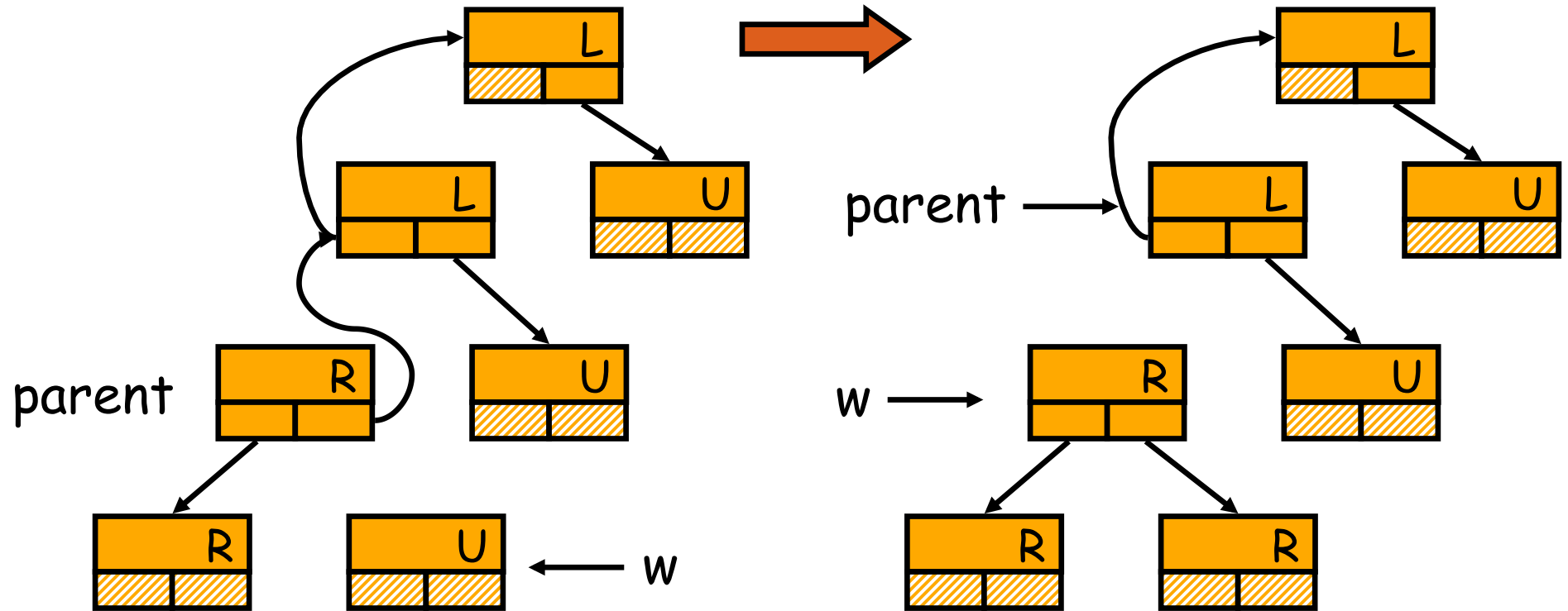
# סימון ללא רקורסיה (המשך)



# סימון ללא רקורסיה (המשך)



# סימון ללא רקורסיה (המשך)



בסוף ריצת הפרוצדורה כל התוויות של האובייקטים הנגישים מסומנות R.

## Pointer Reversal - variable sized nodes

- Two additional variables for each node:
  - i-field - current recursion index.
  - n-field - node's number of children.

## Conclusion (Pointer Reversal)

- ☺ No additional data structure required.
- ☹ Every heap's node contains extra information -  $2\log_2 n$  bit/node.
- ☹ Every node's visit includes several fields' update.

# Conclusion (Mark & Sweep)

- Mark and sweep is a simple method. Advanced versions are common in real systems.
- Lots of engineering tricks are available. (We've seen a few.)
- One of the main obstacles is fragmentation.

Our next topic is [compaction](#).