# DATA TYPES & MEMORY MANAGEMENT

## PRINCIPLES OF PROGRAMMING LANGUAGES


Norbert Zeh

Winter 2018

Dalhousie University

Where subroutines allow us to build control abstractions, a language's type system determines the kind of data abstractions we can build.

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

## ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# TYPE SYSTEMS

## Type System

Mechanism for defining types and associating them with operations that can be performed on objects of each type:

- Built-in types with built-in operations
- Custom operations for built-in and custom types

### Type System

Mechanism for defining types and associating them with operations that can be performed on objects of each type:

- Built-in types with built-in operations
- Custom operations for built-in and custom types

A type system includes rules that specify

- **Type equivalence:** Do two values have the same type? (Structural equivalence vs name equivalence)

### Type System

Mechanism for defining types and associating them with operations that can be performed on objects of each type:

- Built-in types with built-in operations
- Custom operations for built-in and custom types

A type system includes rules that specify

- **Type equivalence:** Do two values have the same type? (Structural equivalence vs name equivalence)
- **Type compatibility:** Can a value of a certain type be used in a certain context?

### Type System

Mechanism for defining types and associating them with operations that can be performed on objects of each type:

- Built-in types with built-in operations
- Custom operations for built-in and custom types

A type system includes rules that specify

- **Type equivalence:** Do two values have the same type? (Structural equivalence vs name equivalence)
- **Type compatibility:** Can a value of a certain type be used in a certain context?
- **Type inference:** How is the type of an expression computed from the types of its parts?

**Strongly typed**

Prohibits the application of an operation to any object not supporting this operation.

### Strongly typed

Prohibits the application of an operation to any object not supporting this operation.

### Statically typed

Strongly typed and type checking is performed at compile time (Pascal, C, Haskell, ...)

# COMMON KINDS OF TYPE SYSTEMS

**Strongly typed**

Prohibits the application of an operation to any object not supporting this operation.

**Statically typed**

Strongly typed and type checking is performed at compile time (Pascal, C, Haskell, ...)

**Dynamically typed**

Strongly typed and type checking is performed at runtime (LISP, Smalltalk, Python, ...)

# COMMON KINDS OF TYPE SYSTEMS

**Strongly typed**

Prohibits the application of an operation to any object not supporting this operation.

**Statically typed**

Strongly typed and type checking is performed at compile time (Pascal, C, Haskell, ...)

**Dynamically typed**

Strongly typed and type checking is performed at runtime (LISP, Smalltalk, Python, ...)

In some statically typed languages (e.g., ML), the programmer does not specify types at all. They are inferred by the compiler.

# DEFINITION OF TYPES

Similar to subroutines in many languages, defining a type has two parts:

- A type's declaration introduces its name into the current scope.
- A type's definition describes the type (the simpler types it is composed of).

Similar to subroutines in many languages, defining a type has two parts:

- A type's declaration introduces its name into the current scope.
- A type's definition describes the type (the simpler types it is composed of).

**Classification of types:**

Similar to subroutines in many languages, defining a type has two parts:

- A type's declaration introduces its name into the current scope.
- A type's definition describes the type (the simpler types it is composed of).

**Classification of types:**

- **Denotational:** A type is a set of values.

Similar to subroutines in many languages, defining a type has two parts:

- A type's declaration introduces its name into the current scope.
- A type's definition describes the type (the simpler types it is composed of).

Classification of types:

- **Denotational:** A type is a set of values.
- **Constructive:** A type is built-in or composite.

# DEFINITION OF TYPES

Similar to subroutines in many languages, defining a type has two parts:

- A type's declaration introduces its name into the current scope.
- A type's definition describes the type (the simpler types it is composed of).

**Classification of types:**

- **Denotational:** A type is a set of values.
- **Constructive:** A type is built-in or composite.
- **Abstration-based:** A type is defined by an interface, the set of operations it supports.

**Built-in types:** Integers, Booleans, characters, “real” numbers, …

**Built-in types:** Integers, Booleans, characters, “real” numbers, ...

**Enumeration and range types:** (Neither built-in nor composite)

- C: `enum t { A, B };`
- Pascal: `0..100`

**Built-in types:** Integers, Booleans, characters, “real” numbers, …

**Enumeration and range types:** (Neither built-in nor composite)

- C: `enum t { A, B };`
- Pascal: `0..100`

**Composite types:** Records, arrays, files, lists, sets, pointers, …

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

A nested record definition in Pascal:

```
type ore = record
  name             : short_string;
  element_yielded : record
    name           : two_chars;
    atomic_n       : integers;
    atomic_weight : real;
    metallic       : Boolean
  end
end;
```

A nested record definition in Pascal:

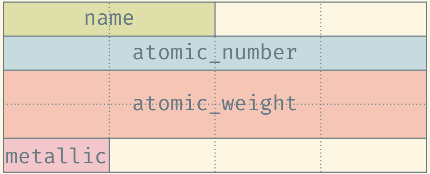
```
type ore = record
  name            : short_string;
  element_yielded : record
    name           : two_chars;
    atomic_n       : integers;
    atomic_weight  : real;
    metallic       : Boolean
  end
end;
```

Accessing fields:

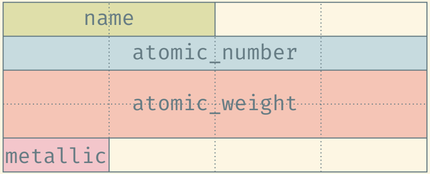
- `ore.element_yielded.name`
- `name of element_yielded of ore`

# MEMORY LAYOUT OF RECORDS

Aligned (fixed ordering):

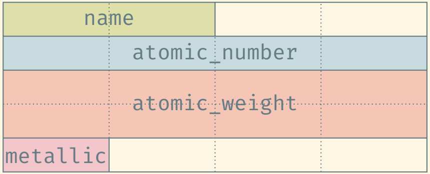


Aligned (fixed ordering):

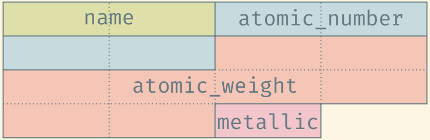



- — Potential waste of space
- + One instruction per element access
- + Guaranteed layout in memory
  (Good for systems programming)

Aligned (fixed ordering):



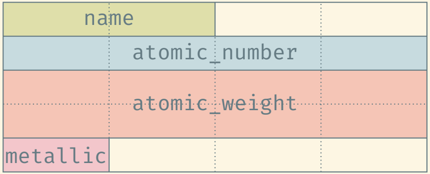

Packed:




— Potential waste of space
+ One instruction per element access
+ Guaranteed layout in memory
(Good for systems programming)

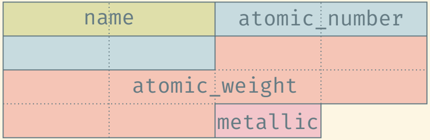
# MEMORY LAYOUT OF RECORDS

Aligned (fixed ordering):




- − Potential waste of space
- + One instruction per element access
- + Guaranteed layout in memory
  (Good for systems programming)

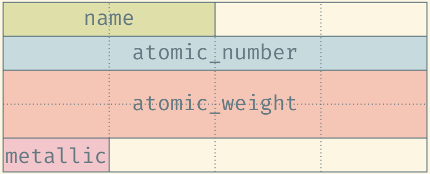
Packed:



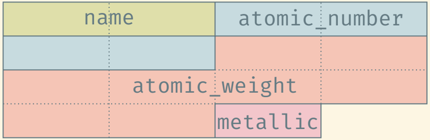

- + No waste of space
- − Multiple instructions per element access
- + Guaranteed layout in memory
  (Good for systems programming)

# MEMORY LAYOUT OF RECORDS

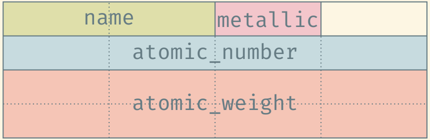
## Aligned (fixed ordering):




## Packed:




## Aligned (optimized ordering):

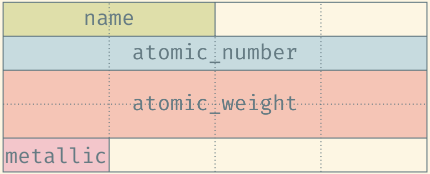



- − Potential waste of space
- \+ One instruction per element access
- \+ Guaranteed layout in memory
  (Good for systems programming)

- \+ No waste of space
- − Multiple instructions per element access
- \+ Guaranteed layout in memory
  (Good for systems programming)

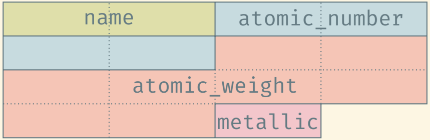
# MEMORY LAYOUT OF RECORDS

Aligned (fixed ordering):



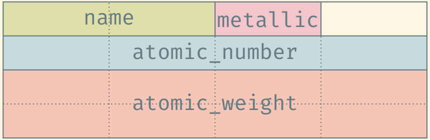

- − Potential waste of space
- + One instruction per element access
- + Guaranteed layout in memory
  (Good for systems programming)

Packed:




- + No waste of space
- − Multiple instructions per element access
- + Guaranteed layout in memory
  (Good for systems programming)

Aligned (optimized ordering):




- ± Reduced space overhead
- + One instruction per element access
- − No guarantee of layout in memory
  (Bad for systems programming)

## ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

## ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

| Stored where? | Shape fixed when? | | |
|---|---|---|---|
| | Compile time | Elaboration time | Dynamic |
| Static address | Static | — | — |
| Stack | Local | Local | — |
| Heap | Dynamic | Dynamic | Dynamic |

Issues:

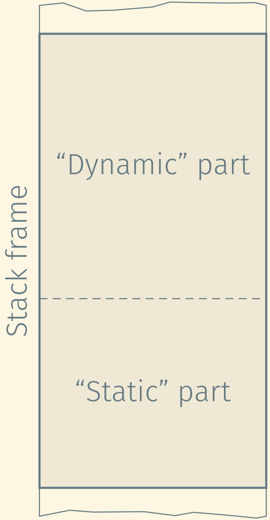
- Memory allocation
- Bounds checks
- Index calculations (higher-dimensional arrays)

# ELABORATION-TIME SHAPE BINDING AND DOPE VECTORS


Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

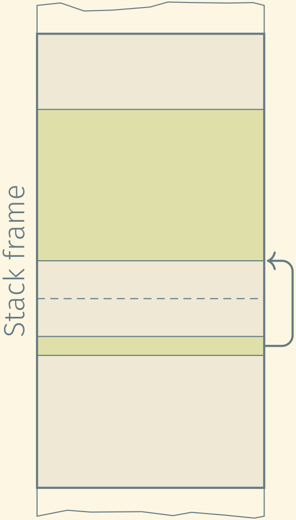
# ELABORATION-TIME SHAPE BINDING AND DOPE VECTORS




Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

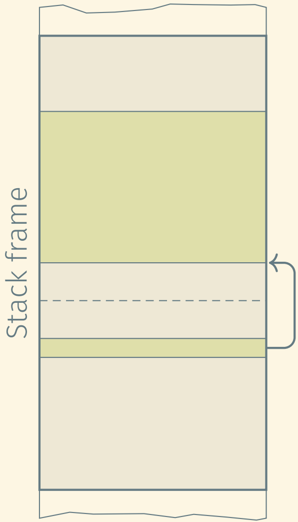
How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

# ELABORATION-TIME SHAPE BINDING AND DOPE VECTORS




Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

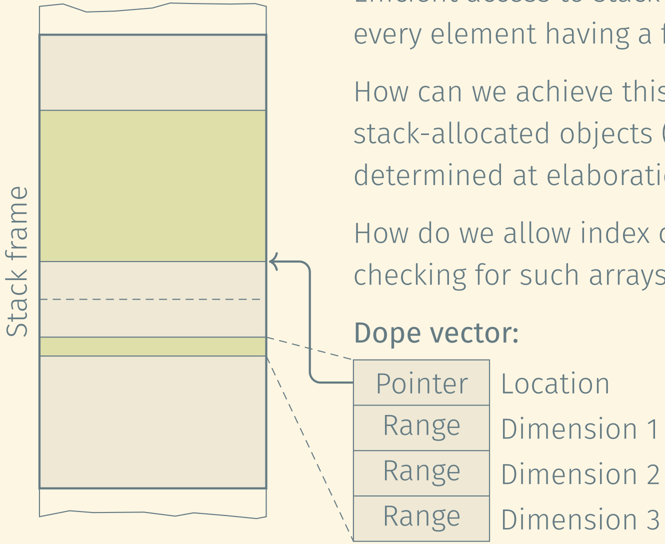
How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?




Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

How do we allow index calculation and bounds checking for such arrays (and heap-allocated arrays)?



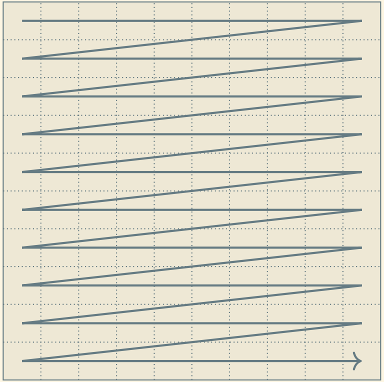

Efficient access to stack-allocated objects is based on every element having a fixed offset in the stack frame.

How can we achieve this when we allow stack-allocated objects (e.g., arrays) to have sizes determined at elaboration time?

How do we allow index calculation and bounds checking for such arrays (and heap-allocated arrays)?

**Dope vector:**

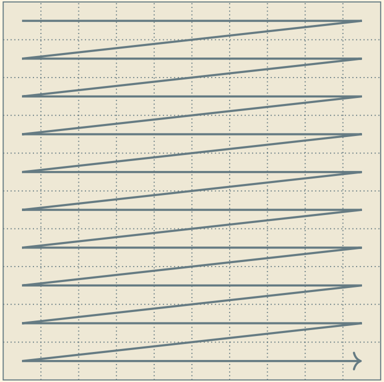
## CONTIGUOUS MEMORY LAYOUT OF 2-D ARRAYS



Row-major layout

Column-major layout

# CONTIGUOUS MEMORY LAYOUT OF 2-D ARRAYS



Row-major layout

Column-major layout

There are more sophisticated block-recursive layouts which, combined with the right algorithms, achieve much better cache efficiency than the above.

# CONTIGUOUS VS ROW-POINTER LAYOUT

```
char days[][10] = {
  "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| M | o | n | d | a | y | \0 | | | |
| T | u | e | s | d | a | y | \0 | | |
| W | e | d | n | e | s | d | a | y | \0 |
| T | h | u | r | s | d | a | y | \0 | |
| F | r | i | d | a | y | \0 | | | |
| S | a | t | u | r | d | a | y | \0 | |
| S | u | n | d | a | y | \0 | | | |

# CONTIGUOUS VS ROW-POINTER LAYOUT

```c
char days[][10] = {
  "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```

```c
char *days[] = {
  "Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday",
  "Sunday"
};
days[2][3] == 's';
```


# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# ASSOCIATIVE ARRAYS

- Allow arbitrary elements as indexes.
- Directly supported in Perl and many other scripting languages.
- C++ and Java call them maps (because they're not really arrays!)
- Scheme calls them association lists (A-lists).

# ASSOCIATIVE ARRAYS

- Allow arbitrary elements as indexes.
- Directly supported in Perl and many other scripting languages.
- C++ and Java call them maps (because they're not really arrays!)
- Scheme calls them association lists (A-lists).

**Example:**
After `(define e '((a 1) (b 2) (c 3)))`, `(assoc 'a e)` returns `(a 1)`.

- Allow arbitrary elements as indexes.
- Directly supported in Perl and many other scripting languages.
- C++ and Java call them maps (because they're not really arrays!)
- Scheme calls them association lists (A-lists).

**Example:**
After `(define e '((a 1) (b 2) (c 3)))`, `(assoc 'a e)` returns `(a 1)`.

**Different lookup operations for different notions of equality:**

- `assq` uses `eq?` (identity)
- `assoc` uses `equal?` (same value)
- `assv` uses `eqv?` (halfway in between)

# ARRAYS, LISTS, AND STRINGS

Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

# ARRAYS, LISTS, AND STRINGS

Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

# ARRAYS, LISTS, AND STRINGS

Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

Functional arrays that allow updates without copying the entire array are seriously non-trivial to implement.

Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

Functional arrays that allow updates without copying the entire array are seriously non-trivial to implement.

Lists are naturally recursive and thus fit extremely well into the recursive approach taken to most problems in functional programming.

Most imperative languages provide excellent built-in support for array manipulation but not for operations on lists.

Most functional languages provide excellent built-in support for list manipulation but not for operations on arrays.

Arrays are a natural way to store sequences when manipulating individual elements in place (i.e., imperatively).

Functional arrays that allow updates without copying the entire array are seriously non-trivial to implement.

Lists are naturally recursive and thus fit extremely well into the recursive approach taken to most problems in functional programming.

Strings are arrays of characters in imperative languages and lists of characters in functional languages.

## ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# POINTERS

- Point to memory locations that store data (often typed, e.g., `int*`).
- Not needed when using a reference model of variables (Lisp, ML, Clu, Java).
- Needed for recursive types when using a value model of variables (C, Pascal, Ada).

# POINTERS

- Point to memory locations that store data (often typed, e.g., `int*`).
- Not needed when using a reference model of variables (Lisp, ML, Clu, Java).
- Needed for recursive types when using a value model of variables (C, Pascal, Ada).

**Storage reclamation:**

- Explicit (manual)
- Automatic (garbage collection)

- Point to memory locations that store data (often typed, e.g., `int*`).
- Not needed when using a reference model of variables (Lisp, ML, Clu, Java).
- Needed for recursive types when using a value model of variables (C, Pascal, Ada).

**Storage reclamation:**

- Explicit (manual)
- Automatic (garbage collection)

**Advantages and disadvantages of explicit storage reclamation:**

+ Garbage collection can incur serious run-time overhead.
– Potential for memory leaks.
– Potential for dangling pointers and segmentation faults.

# POINTER ALLOCATION AND DEALLOCATION

C:

- `p = (element *)malloc(sizeof(element))`
- `free(p)`
- Not type-safe, explicit deallocation

Pascal:

- `new(p)`
- `dispose(p)`
- Type-safe, explicit deallocation

Java/C++:

- `p = new element()` (semantics different between Java and C++, how?)
- `delete p` (in C++)
- Type-safe, explicit deallocation in C++, garbage collection in Java

## ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# DANGLING REFERENCES

**Dangling reference**

A pointer to an already reclaimed object

**Dangling reference**

A pointer to an already reclaimed object

Can only happen when reclamation of objects is the responsibility of the programmer.

# DANGLING REFERENCES

**Dangling reference**

A pointer to an already reclaimed object

Can only happen when reclamation of objects is the responsibility of the programmer.

Dangling references are notoriously hard to debug and a major source of program misbehaviour and security holes.

# DANGLING REFERENCES

**Dangling reference**

A pointer to an already reclaimed object

Can only happen when reclamation of objects is the responsibility of the programmer.

Dangling references are notoriously hard to debug and a major source of program misbehaviour and security holes.

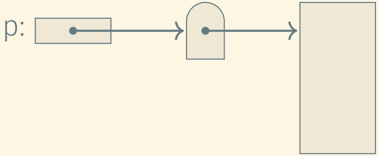
**Techniques to catch them:**

- Tombstones
- Keys and locks

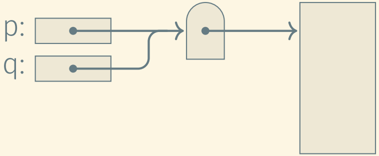
**Idea:** Better crash and expose the bug than do the wrong thing.

# TOMBSTONES (1)

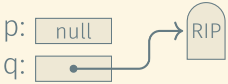
`new(p)`




`q := p`




`delete(p)`



# TOMBSTONES (2)

Issues:

- Space overhead
- Runtime overhead (two cache misses instead of one)
- Check for invalid tombstones = hardware interrupt (cheap):
  - RIP = null pointer
- How to allocate the tombstones?
  - From separate heap (no fragmentation)
  - Need reference count or other garbage collection strategy to determine when I can delete a tombstone.
  - Need to track pointers to objects on the stack, in order to invalidate their tombstones.

# TOMBSTONES (2)

**Issues:**

- Space overhead
- Runtime overhead (two cache misses instead of one)
- Check for invalid tombstones = hardware interrupt (cheap):
  - RIP = null pointer
- How to allocate the tombstones?
  - From separate heap (no fragmentation)
  - Need reference count or other garbage collection strategy to determine when I can delete a tombstone.
  - Need to track pointers to objects on the stack, in order to invalidate their tombstones.

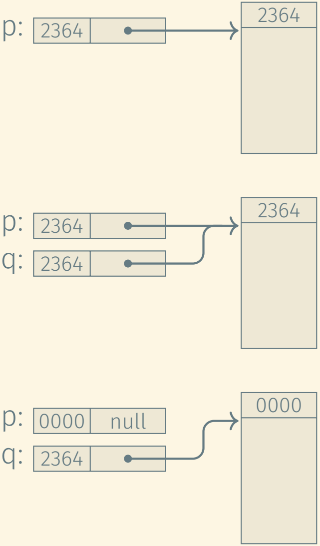
**Interesting side effect:**

- The extra level of indirection allows us to compact memory.

```
q := p
```

```
q := p
```

```
q := p
```



# LOCKS AND KEYS (2)

Comparison to tombstones:

- Can only be used for heap-allocated objects
- Provides only probabilistic protection
- Unclear which one has higher runtime overhead
- Unclear which one has higher space overhead

**Comparison to tombstones:**

- Can only be used for heap-allocated objects
- Provides only probabilistic protection
- Unclear which one has higher runtime overhead
- Unclear which one has higher space overhead

Languages that provide for such checks for dangling references often allow them to be turned on/off using compile-time flags.

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

# ROAD MAP

- Type Systems
- Records
- Arrays
- Associative arrays
- Pointers
- Memory management
- Garbage collection

**Garbage collection**

Automatic reclamation of heap space

- Essential for functional languages
- Popular in modern imperative languages (Java, Python, Ada, Clu, ...)
- Difficult to implement
- Slower than manual reclamation

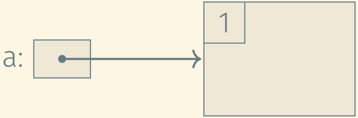
# GARBAGE COLLECTION STRATEGIES

- Reference counts
- Mark and sweep
- Mark and sweep variants:
  - Stop and copy
  - Generational GC

# REFERENCE COUNTS (1)

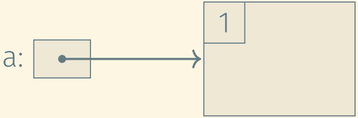
```
a = new Obj();
```

```
a = new Obj();
```

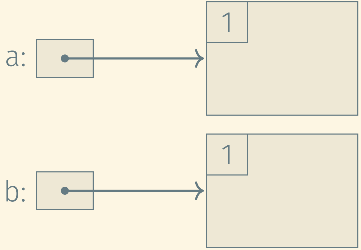


# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
```

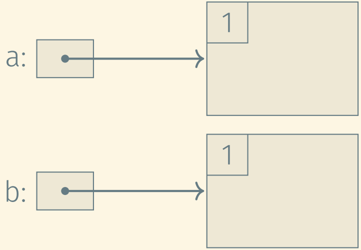


```
a = new Obj();
b = new Obj();
```



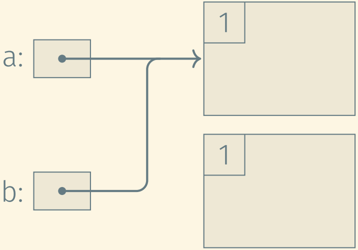
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
```



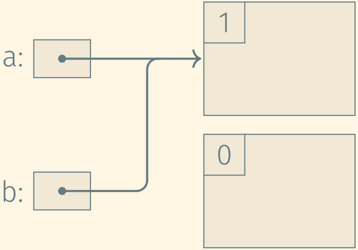
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
```



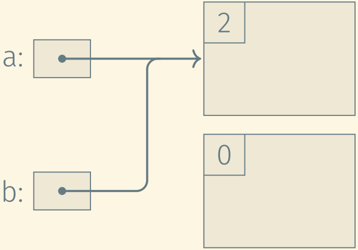
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
```



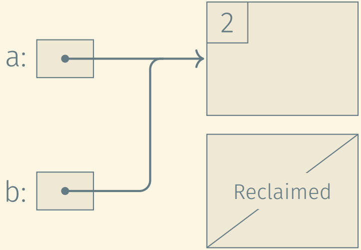
## REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
```



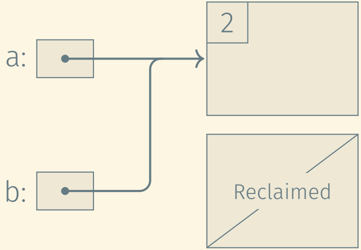
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
```



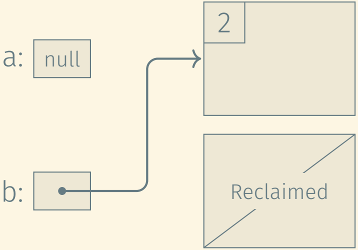
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
a = null;
```



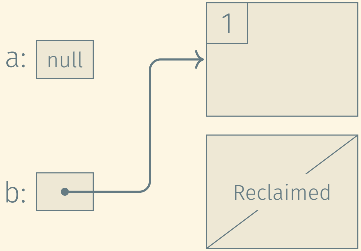
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
a = null;
```



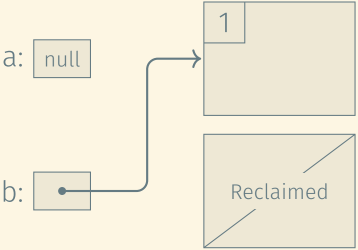
# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
a = null;
```

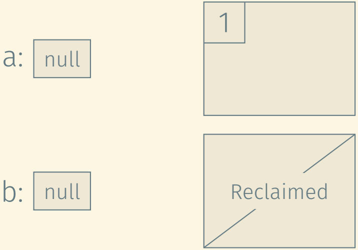


# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```

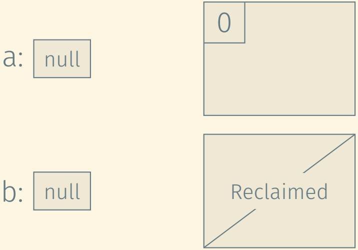


```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```

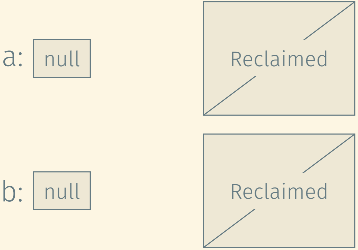


# REFERENCE COUNTS (1)

```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```



```
a = new Obj();
b = new Obj();
b = a;
a = null;
b = null;
```



**Subroutine return:**

- Decrease reference counts for all objects referenced by variables in subroutine's stack frame.
- Requires us to keep track of which entries in a stack frame are pointers.

# REFERENCE COUNTS (2)

**Subroutine return:**

- Decrease reference counts for all objects referenced by variables in subroutine's stack frame.
- Requires us to keep track of which entries in a stack frame are pointers.

**Pros/cons:**

\+ Fairly simple to implement

\+ Fairly low cost

— Does not work when there are circular references. Why?

# MARK AND SWEEP (1)

## Mark and sweep algorithm

- Mark every allocated memory block as useless.
- For every pointer in the static address space and on the stack, mark the block it points to as useful.
- For every block whose status changes from useless to useful, mark the blocks referenced by pointers in this block as useful. Apply this rule recursively.
- Reclaim all blocks marked as useless.

# MARK AND SWEEP (2)

Pros/cons:

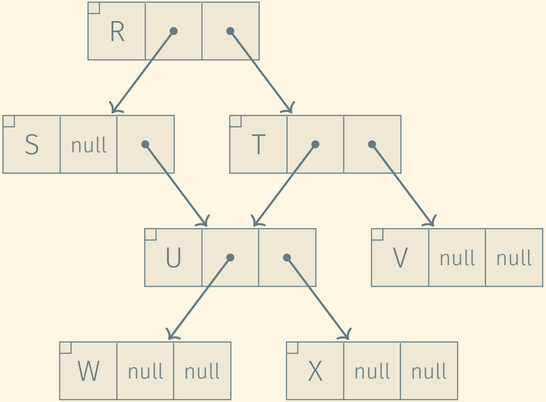
- More complicated to implement.
- Requires inspection of all allocated blocks in a sweep: costly.
- High space usage if the recursion is deep.
- Requires type descriptor at the beginning of each block to know the size of the block and to find the pointers in the block.

# MARK AND SWEEP (2)

Pros/cons:

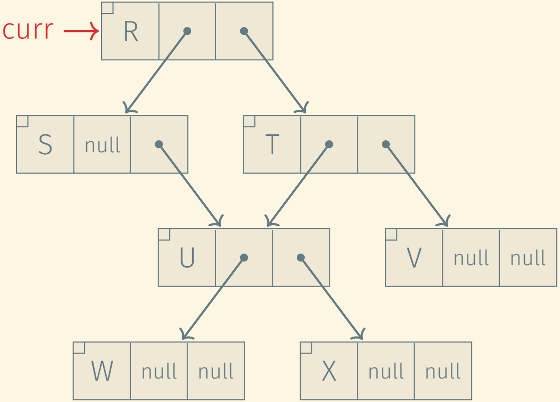
- — More complicated to implement.
- — Requires inspection of all allocated blocks in a sweep: costly.
- — High space usage if the recursion is deep.
- — Requires type descriptor at the beginning of each block to know the size of the block and to find the pointers in the block.
- \+ Works with circular data structures.

# SPACE-EFFICIENT MARK AND SWEEP

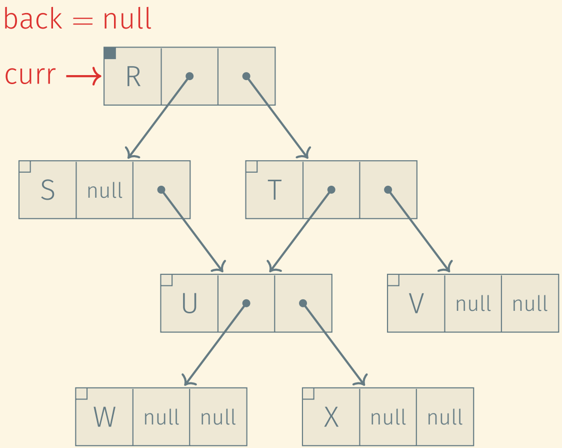


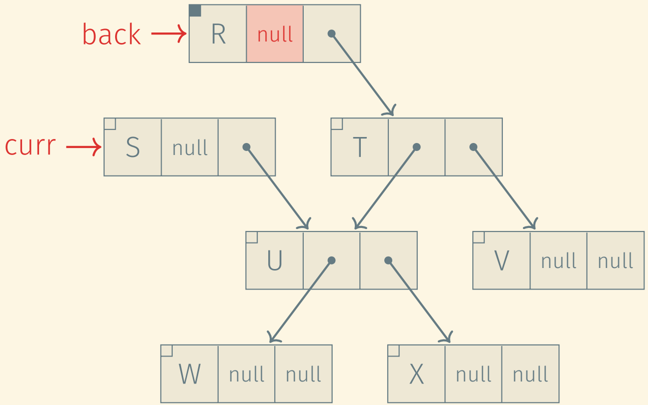
# SPACE-EFFICIENT MARK AND SWEEP

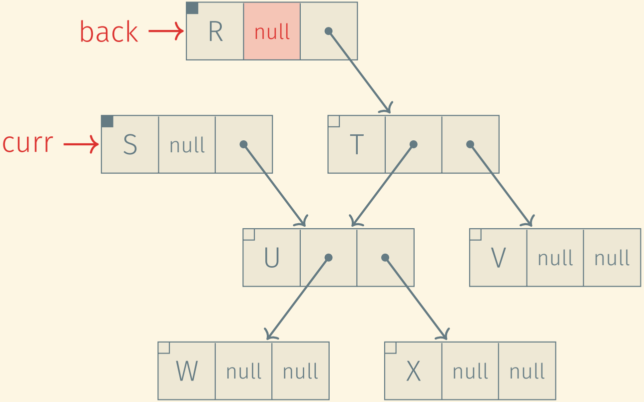
back = null



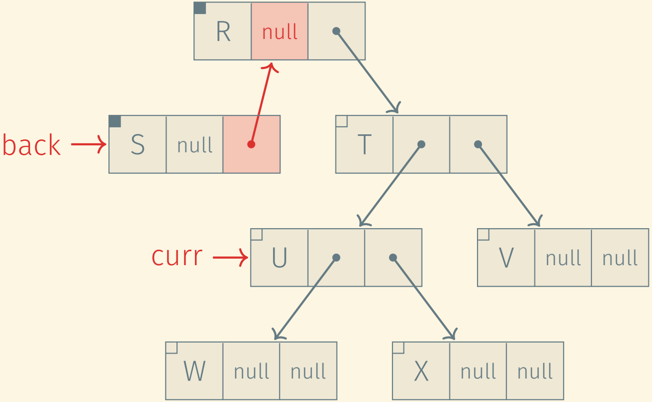
# SPACE-EFFICIENT MARK AND SWEEP

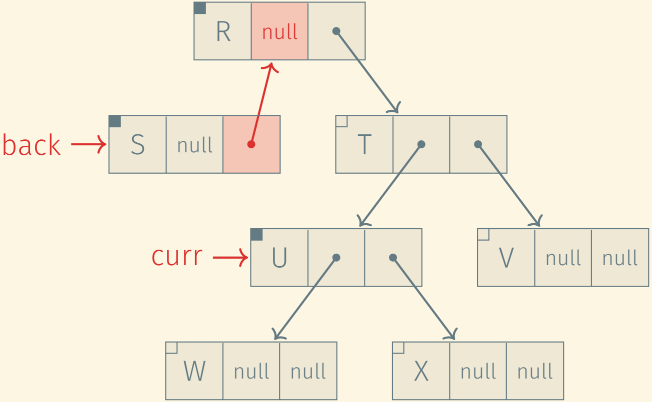


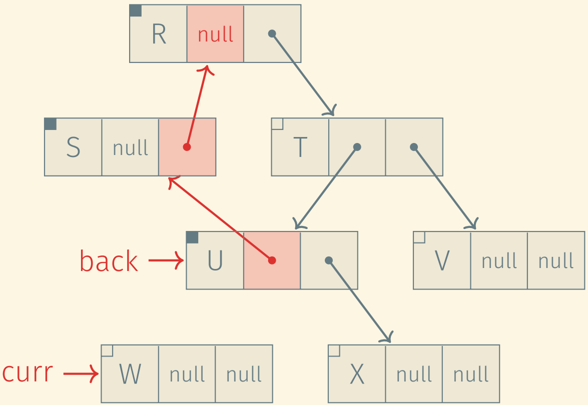

back
R
null
curr
S
null
T
U
V
null
null
W
null
null
X
null
null

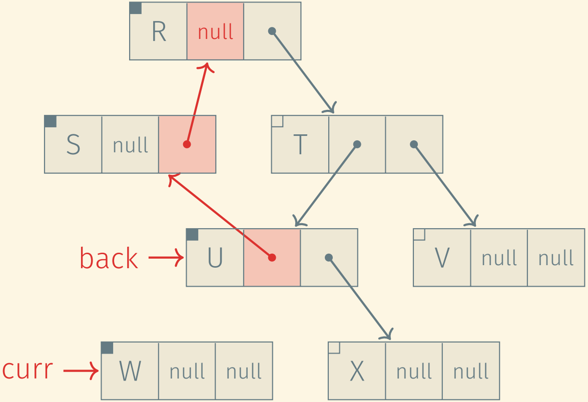

back
R
null
curr
S
null
T
U
V
null
null
W
null
null
X
null
null

# SPACE-EFFICIENT MARK AND SWEEP

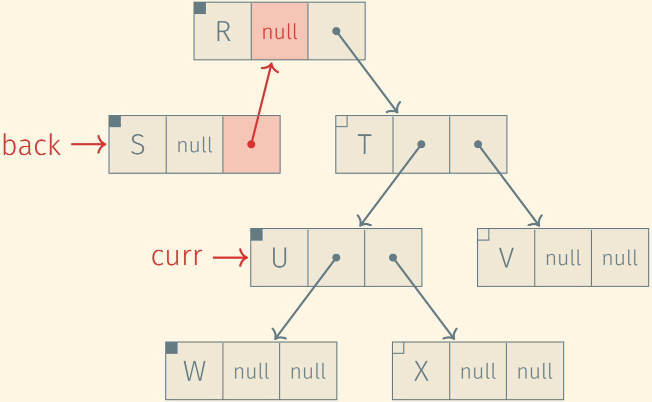



R
null
back
S
null
T
curr
U
V
null
null
W
null
null
X
null
null

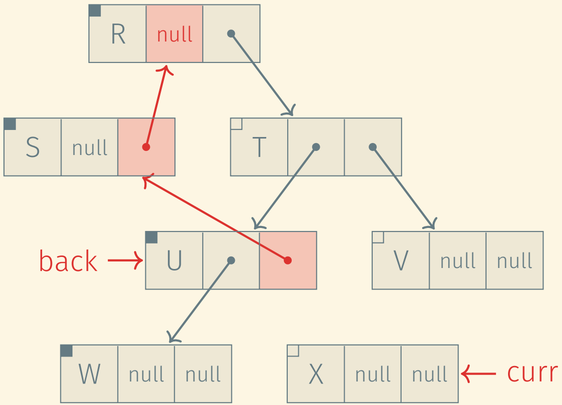

R
null
S
null
T
back
U
V
null
null
curr
W
null
null
X
null
null

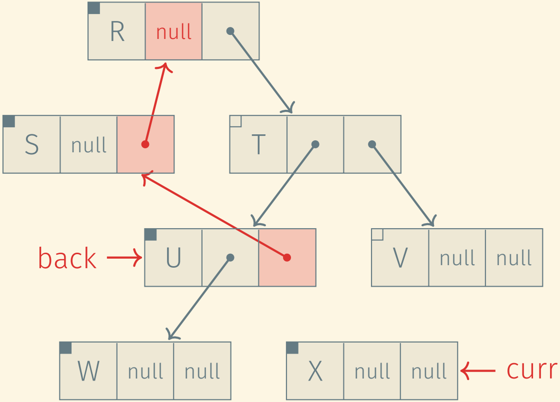

R
null
S
null
T
back
U
V
null
null
curr
W
null
null
X
null
null

# SPACE-EFFICIENT MARK AND SWEEP

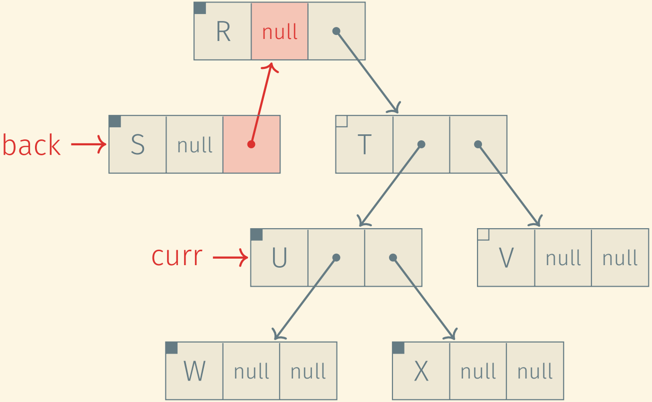


# SPACE-EFFICIENT MARK AND SWEEP

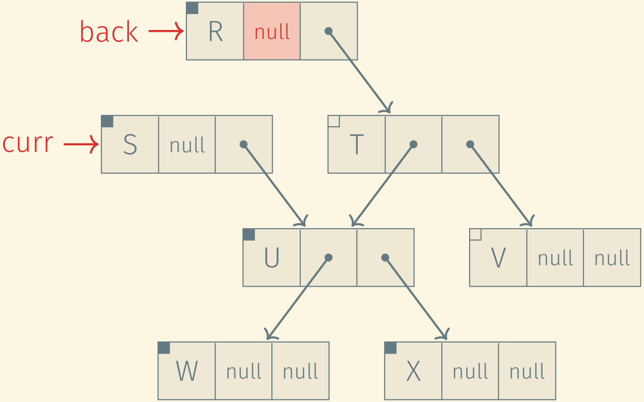



R
null
S
null
T
back
U
V
null
null
W
null
null
X
null
null
curr

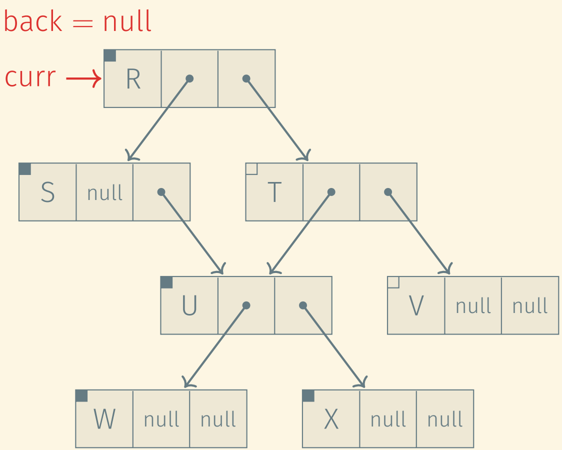
# SPACE-EFFICIENT MARK AND SWEEP



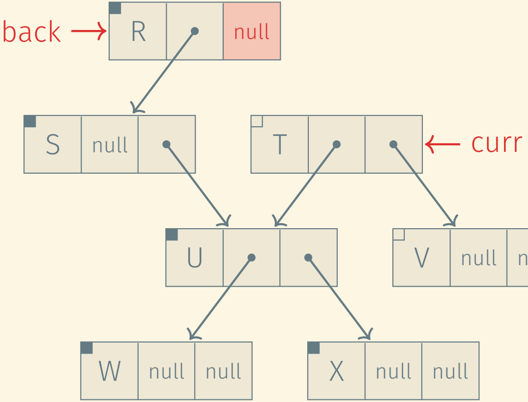
# SPACE-EFFICIENT MARK AND SWEEP



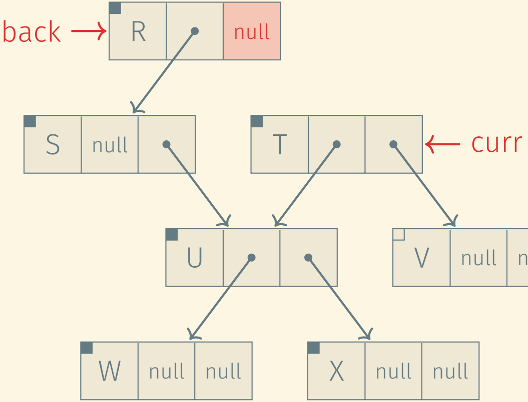
# SPACE-EFFICIENT MARK AND SWEEP



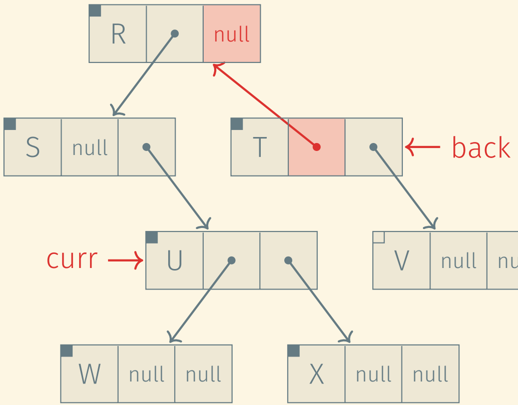
# SPACE-EFFICIENT MARK AND SWEEP

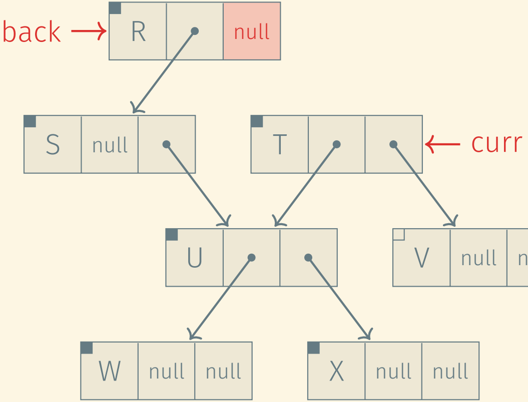


# SPACE-EFFICIENT MARK AND SWEEP

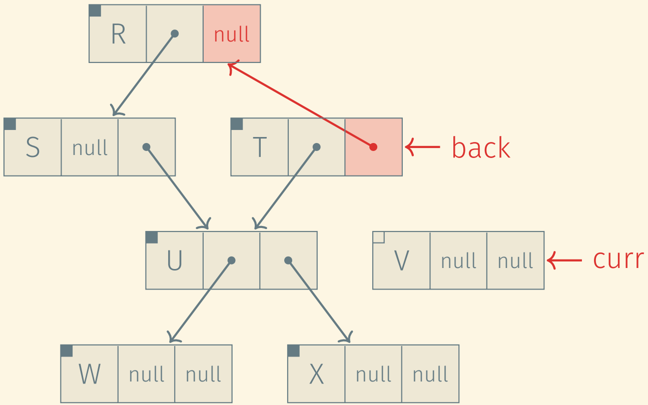


# SPACE-EFFICIENT MARK AND SWEEP

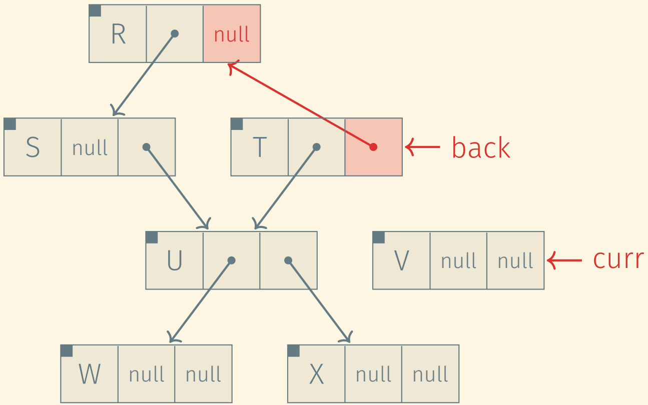



back
R
null
S
null
T
curr
U
V
null
null
W
null
null
X
null
null

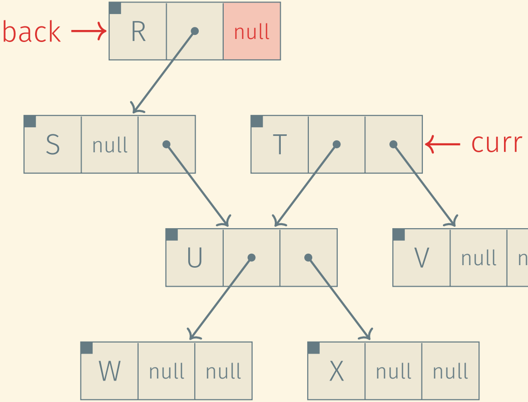
# SPACE-EFFICIENT MARK AND SWEEP



# SPACE-EFFICIENT MARK AND SWEEP

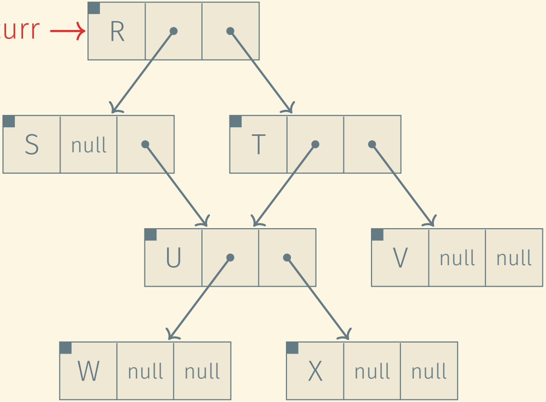


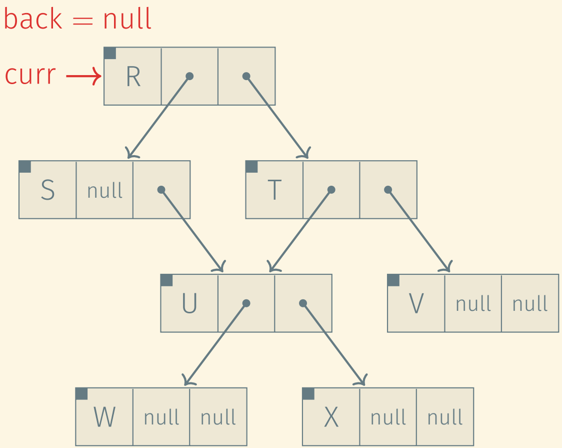
# SPACE-EFFICIENT MARK AND SWEEP



back = null

curr →






**Space:** $1 + \lceil \log_2 f \rceil$ bits per record ($f$ = number of fields in record)

# MARK AND SWEEP VARIANT: STOP AND COPY (1)

## Stop and copy algorithm

- Heap is divided into two halves. Allocation happens in the first half.
- Once the first half is full, start garbage collection and compaction:
    - Find useful objects as in standard mark and sweep but without first marking objects as useless.
    - Copy every useful object to the second half of the heap and replace it with a tag pointing to the new location.
    - Replace every subsequent pointer to this object with a pointer to the new copy.
    - Swap the roles of the two halves.

Pros/cons:

+ Time proportional to the number of useful objects, not total number of objects.
+ Eliminates external fragmentation.
± Only half the heap is available for allocation. Not really an issue if we have virtual memory.

# MARK AND SWEEP VARIANT: GENERATIONAL GARBAGE COLLECTION (1)

## Generational garbage collection

- Heap is divided into several (often two) regions.
- Allocation happens in the first region.
- Garbage collection:
    - Apply mark and sweep to the first region.
    - Promote every object that survives a small number (often one) of rounds of garbage collection in a region to the next region in a manner similar to stop and copy.
    - Inspect (mark and sweep) subsequent regions only if collection in the regions inspected so far did not free up enough space.

## MARK AND SWEEP VARIANT: GENERATIONAL GARBAGE COLLECTION (1)

### Generational garbage collection

- Heap is divided into several (often two) regions.
- Allocation happens in the first region.
- Garbage collection:
  - Apply mark and sweep to the first region.
  - Promote every object that survives a small number (often one) of rounds of garbage collection in a region to the next region in a manner similar to stop and copy.
  - Inspect (mark and sweep) subsequent regions only if collection in the regions inspected so far did not free up enough space.

**Idea:** Most objects are short-lived. Collection mostly inspects only the first region and is thus cheaper.

# MARK AND SWEEP VARIANT: GENERATIONAL GARBAGE COLLECTION (1)

Two main issues:

- How to discover useful objects in a region without searching the other regions?
- How to update pointers from older regions to younger regions when promoting objects?

# MARK AND SWEEP VARIANT: GENERATIONAL GARBAGE COLLECTION (1)

Two main issues:

- How to discover useful objects in a region without searching the other regions?
- How to update pointers from older regions to younger regions when promoting objects?

Techniques:

- Disallow pointers from old regions to young regions by moving objects around appropriately.
- Keep list of old-to-new pointers. (Requires instrumentation $\Rightarrow$ runtime overhead.)

# CONSERVATIVE GARBAGE COLLECTION

Normally, garbage collectors need to know which positions in an object are pointers. This requires a type descriptor to be stored with each object (space overhead).

# CONSERVATIVE GARBAGE COLLECTION

Normally, garbage collectors need to know which positions in an object are pointers. This requires a type descriptor to be stored with each object (space overhead).

**Conservative garbage collection**

Assume every word is a pointer and mark every memory block referenced by such a “pointer” as useful.

# CONSERVATIVE GARBAGE COLLECTION

Normally, garbage collectors need to know which positions in an object are pointers. This requires a type descriptor to be stored with each object (space overhead).

**Conservative garbage collection**

Assume every word is a pointer and mark every memory block referenced by such a “pointer” as useful.

This may fail to reclaim useless objects because some integer or other value happens to equal the address of the object when interpreted as a pointer.

# CONSERVATIVE GARBAGE COLLECTION

Normally, garbage collectors need to know which positions in an object are pointers. This requires a type descriptor to be stored with each object (space overhead).

**Conservative garbage collection**

Assume every word is a pointer and mark every memory block referenced by such a "pointer" as useful.

This may fail to reclaim useless objects because some integer or other value happens to equal the address of the object when interpreted as a pointer.

Statistically, this is rare.

## SUMMARY

- A language's type system determines the data abstractions we can build.
- Common types include records, arrays, lists (built from records), pointers, ...
- Pointers to heap-allocated objects require us to manage these objects.
- Garbage collection relieves us of the need to explicitly manage heap-allocated objects but has a runtime cost and is non-trivial.