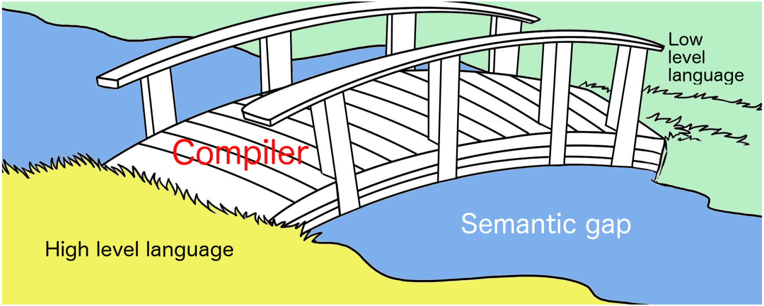
Compilers and computer architecture: Garbage collection

Martin Berger [1]

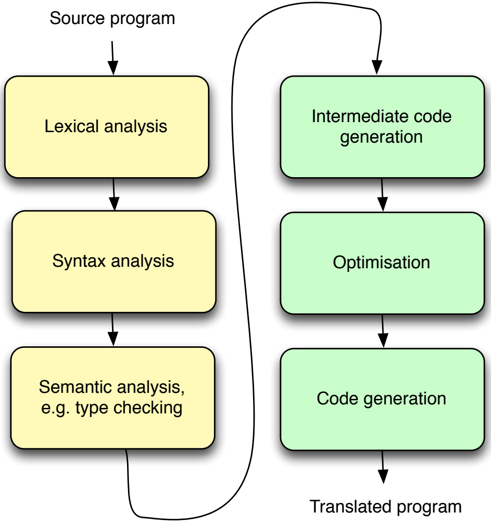
December 2019

[1]Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Recall the structure of compilers



"There is nothing difficult in GC, except to get it to run fast. That's 30-40 years of research." J. Vitek, personal communication, 2016.

What is the question GC is the answer to?

Memory management

Consider the following Java fragment

```
while(serverRunning) {
    NetConnection conn = new NetConnection( ... );
    Customer cust = new Customer(conn);
    cust.act();
    if( ... ) serverRunning = false;
}
```

Memory management

Consider the following Java fragment

```
while(serverRunning) {
   NetConnection conn = new NetConnection( ... );
   Customer cust = new Customer(conn);
   cust.act();
   if( ... ) serverRunning = false;
}
```

Say a `NetConnection` object and a `Customer` object together take 100 KBytes in (heap) memory (realistic for many applications). Say we have 16 GBytes memory available. How many times can we go through the loop before running out of memory (assuming we do nothing to reclaim memory)?

Memory management

Consider the following Java fragment

```
while(serverRunning) {
   NetConnection conn = new NetConnection( ... );
   Customer cust = new Customer(conn);
   cust.act();
   if( ... ) serverRunning = false;
}
```

Say a `NetConnection` object and a `Customer` object together take 100 KBytes in (heap) memory (realistic for many applications). Say we have 16 GBytes memory available. How many times can we go through the loop before running out of memory (assuming we do nothing to reclaim memory)?

Approx. 167772 times. That's not a lot (think Facebook, Amazon, Google).

Memory management

Consider the following Java fragment

```
while(serverRunning) {
   NetConnection conn = new NetConnection( ... );
   Customer cust = new Customer(conn);
   cust.act();
   if( ... ) serverRunning = false;
}
```

Say a `NetConnection` object and a `Customer` object together take 100 KBytes in (heap) memory (realistic for many applications). Say we have 16 GBytes memory available. How many times can we go through the loop before running out of memory (assuming we do nothing to reclaim memory)?

Approx. 167772 times. That's not a lot (think Facebook, Amazon, Google). What should happen then?

(See `prog/IntroExample.java`)

Memory management

Memory management

Consider the following Java fragment

```
while(serverRunning) {
   NetConnection conn = new NetConnection( ... );
   Customer cust = new Customer(conn);
   cust.act();
   if( ... ) serverRunning = false;
}
```

But at the end of each loop iteration, the allocated `conn` and `cust` are no longer usable (static scoping!)

So the heap storage they occupy can be reused!

Memory management

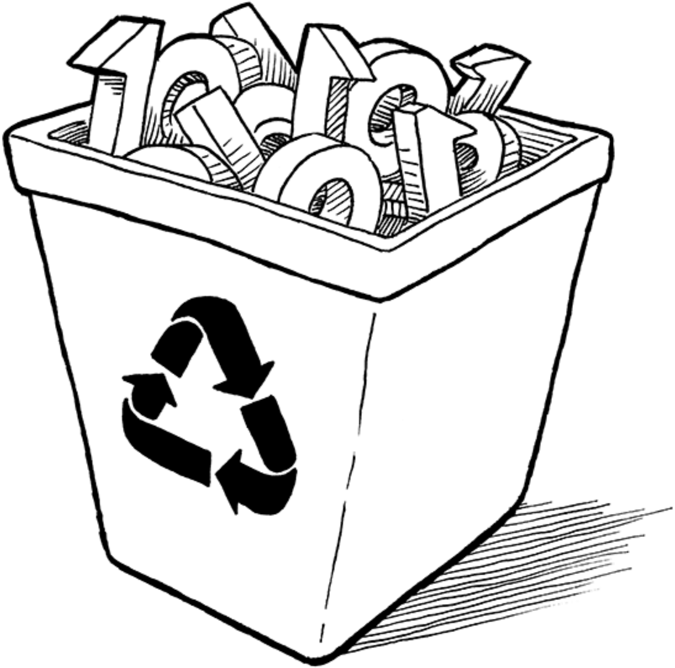
Consider the following Java fragment

```
while(serverRunning) {
   NetConnection conn = new NetConnection( ... );
   Customer cust = new Customer(conn);
   cust.act();
   if( ... ) serverRunning = false;
}
```

But at the end of each loop iteration, the allocated `conn` and `cust` are no longer usable (static scoping!)

So the heap storage they occupy can be reused!

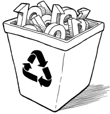
How do we heap reuse storage?



Reusing storage

There are three ways of reusing heap storage.

Reusing storage



There are three ways of reusing heap storage.

- **By hand**: the programmer has to insert commands that reclaim storage. Used in C/C++.

Reusing storage

There are three ways of reusing heap storage.

- **By hand**: the programmer has to insert commands that reclaim storage. Used in C/C++.
- Automatically, using a **garbage collector**. Used in most other modern languages. Java was the first mainstream language to do this, although the concept is much older.

Reusing storage

There are three ways of reusing heap storage.

- **By hand**: the programmer has to insert commands that reclaim storage. Used in C/C++.
- Automatically, using a **garbage collector**. Used in most other modern languages. Java was the first mainstream language to do this, although the concept is much older.
- With the help from the **typing system**. That's Rust's approach. It's brand new. Not currently used in any mainstream language, but might be popular in the future.

Reusing storage

There are three ways of reusing heap storage.

- **By hand**: the programmer has to insert commands that reclaim storage. Used in C/C++.
- Automatically, using a **garbage collector**. Used in most other modern languages. Java was the first mainstream language to do this, although the concept is much older.
- With the help from the **typing system**. That's Rust's approach. It's brand new. Not currently used in any mainstream language, but might be popular in the future.

Let's look at the first two in turn.

Manually reusing storage

In C/C++-like languages we would have to write something like this:

```
while(serverRunning) {
   NetConnection conn = new NetConnection( ... );
   Customer cust = new Customer(conn);
   cust.act();
   if( ... ) serverRunning = false;
   free(cust);
   free(conn);
}
```

To understand what `free` is really doing, let's look at (a simplified model of) what `new` does.

Heap management

Remember we need a heap because some things in the memory outlive procedure/method activations, e.g. this:

```
public Thing doStuff() {
   Thing thing  = new Thing();
   ...
   return thing;
}
```

We cannot store `thing` the activation record of `doStuff`, because `thing` might be used after `doStuff` has returned, and ARs are removed from the stack when the method returns.

Heap management

Remember we need a heap because some things in the memory outlive procedure/method activations, e.g. this:

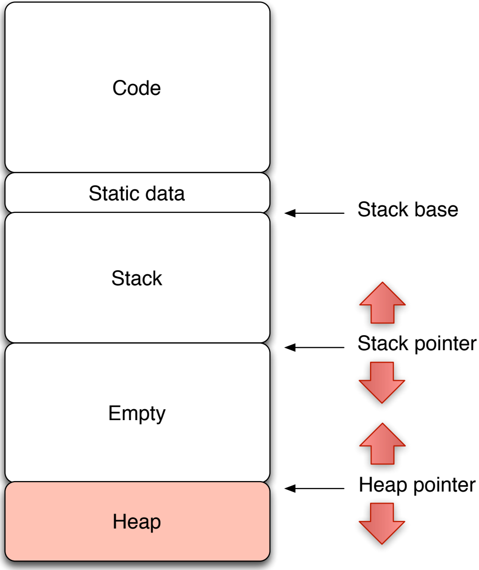
```
public Thing doStuff() {
   Thing thing  = new Thing();
   ...
   return thing;
}
```

We cannot store `thing` the activation record of `doStuff`, because `thing` might be used after `doStuff` has returned, and ARs are removed from the stack when the method returns.

We use the **heap** for such long-lived objects.

Please remember that the concept of heap in compilers has nothing to do with heaps you learn about in algorithms/data structures.

Heap management

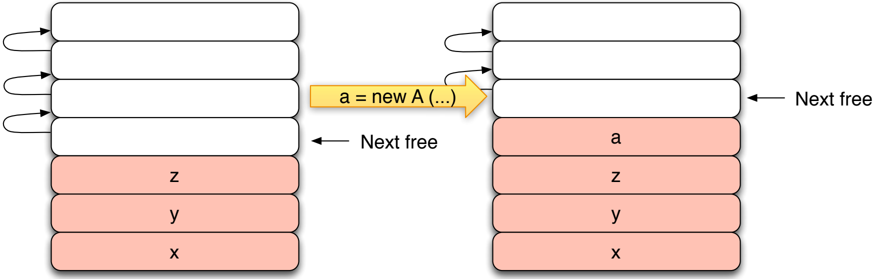


We use the **heap** for such long-lived objects.

Let's look at a very simplified model of heap management.

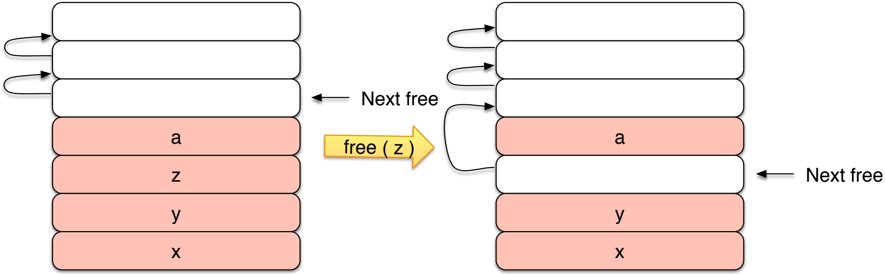
Heap management: allocating memory

Here is a simplified picture of what happens when we allocate memory on the heap, e.g. by `a = new A (...)`.



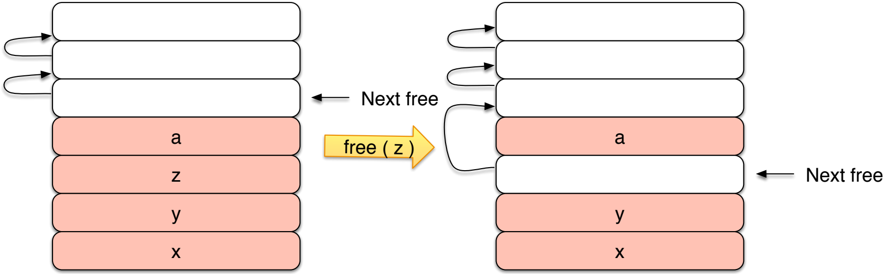
Heap management: freeing memory

Here is a simplified picture of what happens when we free memory on the heap, e.g. by `free(z)`.



Heap management: freeing memory

Here is a simplified picture of what happens when we free memory on the heap, e.g. by `free(z)`.



Note that this is a simplification, e.g. heap allocated memory is not always of the same size.

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

- Forgetting to free memory, leading to **memory leaks.**

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

- Forgetting to free memory, leading to **memory leaks**.
- Accidentally freeing memory more than once.

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

- Forgetting to free memory, leading to **memory leaks**.
- Accidentally freeing memory more than once.
- Using memory after it has been freed, the dreaded **null-pointer dereference** in C/C++.

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

- Forgetting to free memory, leading to **memory leaks**.
- Accidentally freeing memory more than once.
- Using memory after it has been freed, the dreaded **null-pointer dereference** in C/C++.
- Using memory after it has been freed, and reallocated (to something of a different type).

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

- Forgetting to free memory, leading to **memory leaks.**
- Accidentally freeing memory more than once.
- Using memory after it has been freed, the dreaded **null-pointer dereference** in C/C++.
- Using memory after it has been freed, and reallocated (to something of a different type).

These bugs tend to be really hard to find, because cause and effect can be far removed from each other.

Manual heap management

In older languages (C/C++) the programmer manages the heap explicitly, i.e. calls `free` to release memory from the heap, and make it available for other usage. This is problematic:

- Forgetting to free memory, leading to **memory leaks**.
- Accidentally freeing memory more than once.
- Using memory after it has been freed, the dreaded **null-pointer dereference** in C/C++.
- Using memory after it has been freed, and reallocated (to something of a different type).

These bugs tend to be really hard to find, because cause and effect can be far removed from each other.

Fear of the above problems leading to **defensive programming** which can be inefficient and exhibit awkward software style.

Manual heap management: problems

Here is an example of using memory after it has been freed, and reallocated.

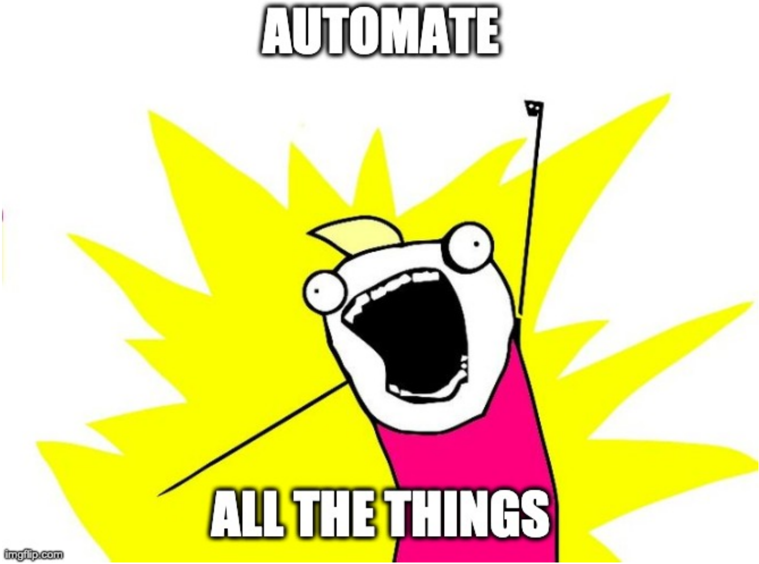
```
a = new A (); // a stored starting at memory
              // cell 1234
...
free(a);
b = new B(); // now b occupies cell 1234
a.f();       // might use memory cell 1234
             // as if it still contained
             // something of type A
```

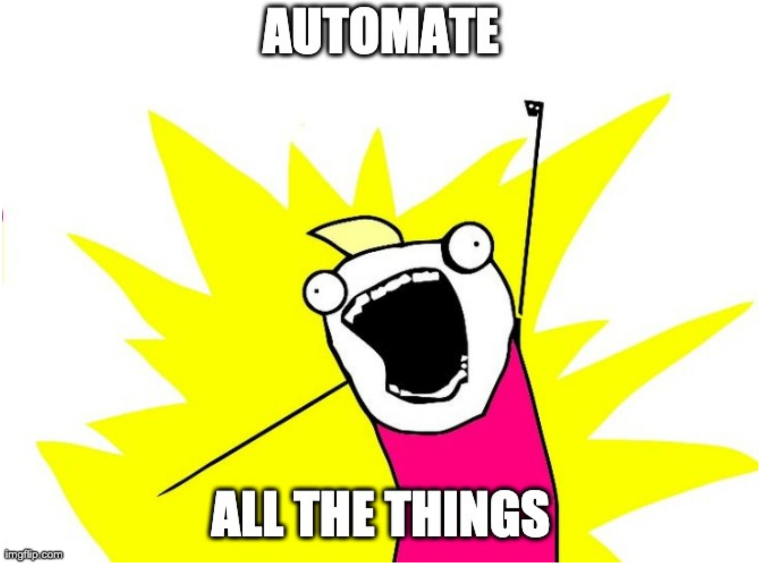
Manual heap management: problems

Here is an example of using memory after it has been freed, and reallocated.

```
a = new A (); // a stored starting at memory
              // cell 1234
...
free(a);
b = new B(); // now b occupies cell 1234
a.f();       // might use memory cell 1234
             // as if it still contained
             // something of type A
```

What can we do about this?


AUTOMATE
ALL THE THINGS
imgflip.com



Memory management is tedious, so why not let the computer do it?

Automatic heap management

This is an old problem, and has been studied since at least the 1950s (LISP). There are two ways of doing this!

Automatic heap management

This is an old problem, and has been studied since at least the 1950s (LISP). There are two ways of doing this!

Type-based

GC (Garbage collection)

Rust & type-based memory management. (Not exam relevant)

Rust & type-based memory management. (Not exam relevant)

Rust seeks to combine the advantages of C/C++ (speed of memory management) with those of GC (safety), by letting the typing system check for absence of memory errors. Problem: complicates language.

Rust & type-based memory management. (Not exam relevant)

Rust seeks to combine the advantages of C/C++ (speed of memory management) with those of GC (safety), by letting the typing system check for absence of memory errors. Problem: complicates language.

Rust is fairly new and is only now (2019) hitting the mainstream.

GC

First mainstream language to use it was Java, where it was introduced because manual heap management in C/C++ caused so many problems.

GC

First mainstream language to use it was Java, where it was introduced because manual heap management in C/C++ caused so many problems.

Originally GC was slow, and resented for that reason. But by now GC is **typically** almost as fast as manual memory management, but **much safer.** (There are edge cases where GC can be much slower.)

GC speed

"There is nothing difficult in GC, except to get it to run fast. That's 30-40 years of research." J. Vitek, personal communication, 2016.

GC speed

"There is nothing difficult in GC, except to get it to run fast. That's 30-40 years of research." J. Vitek, personal communication, 2016.

Fortunately, we've spent those 30-40 years, and you can reap the benefits!

Automatic heap management

GC becoming mainstream is probably the single biggest programming language improvement (in the sense of reducing bugs) in the last two decades.

Garbage collection

```
int [] a = new int[] { 1,2,3 };
a = new int[] { 9,8,7,6,5,4,3,2 };
```

What do we know about the memory allocated in the first line after the last line is executed?

Garbage collection

```
int [] a = new int[] { 1,2,3 };
a = new int[] { 9,8,7,6,5,4,3,2 };
```

What do we know about the memory allocated in the first line after the last line is executed?

The memory allocated in the first line is no longer **reachable** and usable by the program. (Why?)

Garbage collection

```
int [] a = new int[] { 1,2,3 };
a = new int[] { 9,8,7,6,5,4,3,2 };
```

What do we know about the memory allocated in the first line after the last line is executed?

The memory allocated in the first line is no longer **reachable** and usable by the program. (Why?)

Memory that is no longer reachable cannot affect the remainder of the program's computation.

Garbage collection

```
int [] a = new int[] { 1,2,3 };
a = new int[] { 9,8,7,6,5,4,3,2 };
```

What do we know about the memory allocated in the first line after the last line is executed?

The memory allocated in the first line is no longer **reachable** and usable by the program. (Why?)

Memory that is no longer reachable cannot affect the remainder of the program's computation.

Hence: memory that is no longer reachable can be reused.

Garbage collection

The basic idea of GC at a given point in a program's execution is simple, in order to reclaim heap memory:

Garbage collection

The basic idea of GC at a given point in a program's execution is simple, in order to reclaim heap memory:

- Find all the memory that is reachable (from the live variables visible as the given point).

Garbage collection

The basic idea of GC at a given point in a program's execution is simple, in order to reclaim heap memory:

- Find all the memory that is reachable (from the live variables visible as the given point).
- All memory that is not reachable, can be used for allocation (is free). We also say that non-reachable memory (at the given point in a program's execution) is **garbage.**

Garbage collection

The basic idea of GC at a given point in a program's execution is simple, in order to reclaim heap memory:

- Find all the memory that is reachable (from the live variables visible as the given point).
- All memory that is not reachable, can be used for allocation (is free). We also say that non-reachable memory (at the given point in a program's execution) is **garbage.**

The details of how to implement this basic idea can vary a great deal and strongly influence how well a GC works.

Garbage collection

The basic idea of GC at a given point in a program's execution is simple, in order to reclaim heap memory:

- Find all the memory that is reachable (from the live variables visible as the given point).
- All memory that is not reachable, can be used for allocation (is free). We also say that non-reachable memory (at the given point in a program's execution) is **garbage**.

The details of how to implement this basic idea can vary a great deal and strongly influence how well a GC works.

Before we can study GCs in more detail, we must be more precise what it means for memory to be reachable by a program.

Reachability

The big question in GC: how can we know if a piece of memory is free (available for reallocation), or is still being used by the program and cannot (at this point) be reallocated?

Reachability

The big question in GC: how can we know if a piece of memory is free (available for reallocation), or is still being used by the program and cannot (at this point) be reallocated?

Let's look at an example.

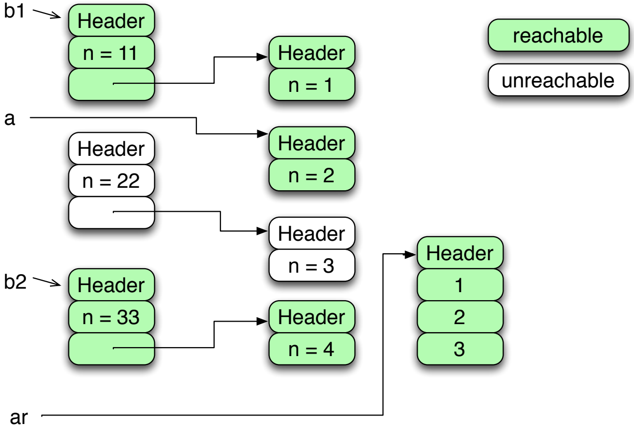
Reachability

```
class A {
    public int n;
    public A(int n) { this.n = n; } }

class B {
    public int n;
    public A a;
    public B(int n, A a) {
        this.n = n;
        this.a = a; } }
...
    public static void main ( String [] args ) {
        A a  = new A ( 1 );
        B b1 = new B ( 11, a );
          a  = new A ( 2 );
        B b2 = new B ( 22, new A ( 3 ) );
          b2 = new B ( 33, new A ( 4 ) );
        int[] ar = new int[] { 1,2,3 };
        ...
```

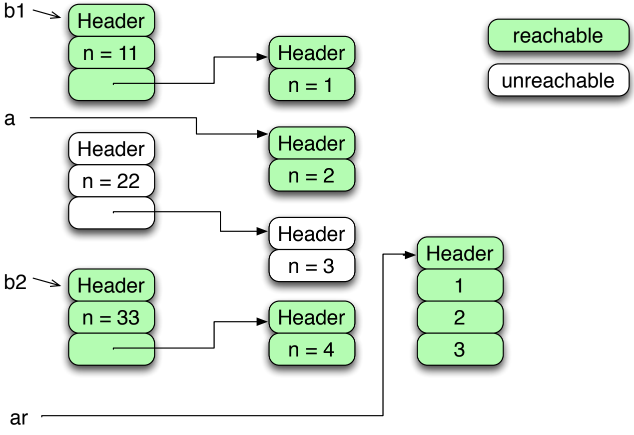
Reachability

The picture below shows the heap just after the array `ar` has been allocated.



Reachability

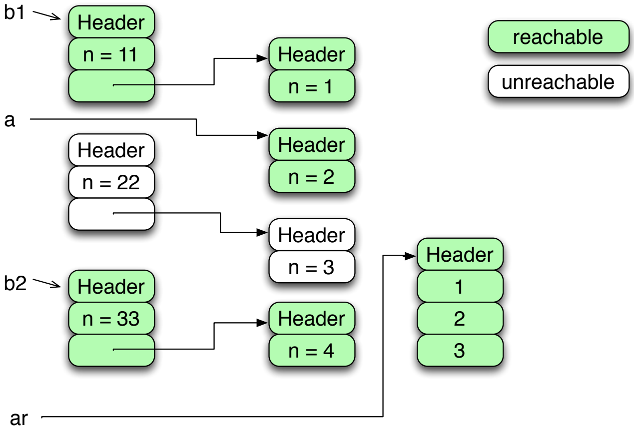
The picture below shows the heap just after the array `ar` has been allocated.



Some cells are reachable directly from the variables `b1`, `a`, `b2` and `ar`. But some are reachable only indirectly through **embedded pointers.**

Reachability

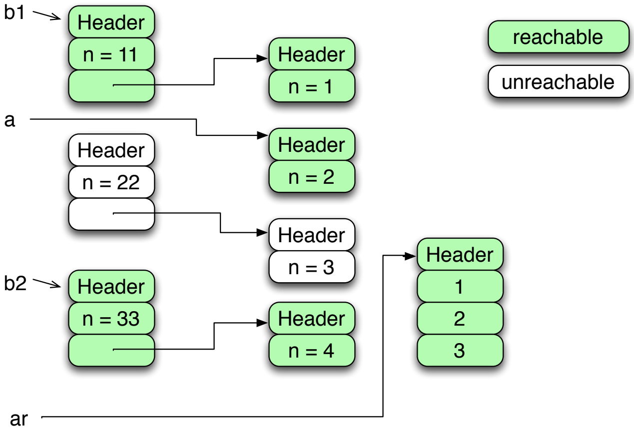
The picture below shows the heap just after the array `ar` has been allocated.



Question: why don't we have to worry about the pointers to the method tables in the headers?

Reachability

The picture below shows the heap just after the array `ar` has been allocated.



Answer: methods are statically allocated. They don't change during program execution (in Java).

Reachability

The key idea of **reachability** is this, for any point in a program we can work out which variables are **live** as follows:

Reachability

The key idea of **reachability** is this, for any point in a program we can work out which variables are **live** as follows:

- We look at the program syntax (known at compile time) to see which variables are in scope. These variables are called **roots**.

Reachability

The key idea of **reachability** is this, for any point in a program we can work out which variables are **live** as follows:

- We look at the program syntax (known at compile time) to see which variables are in scope. These variables are called **roots**.
- If a root is a pointer to an object on the heap, (e.g. the memory `a` points to in `a = new A(...)`, or in `a = new int [5]`, then that object is reachable.

Reachability

The key idea of **reachability** is this, for any point in a program we can work out which variables are **live** as follows:

- We look at the program syntax (known at compile time) to see which variables are in scope. These variables are called **roots**.
- If a root is a pointer to an object on the heap, (e.g. the memory `a` points to in `a = new A(...)`, or in `a = new int [5]`, then that object is reachable.
- If an object on the heap is reachable and contains a pointer to another object in the heap, then the second object is also reachable.

Reachability

The key idea of **reachability** is this, for any point in a program we can work out which variables are **live** as follows:

- We look at the program syntax (known at compile time) to see which variables are in scope. These variables are called **roots**.
- If a root is a pointer to an object on the heap, (e.g. the memory `a` points to in `a = new A(...)`, or in `a = new int [5]`, then that object is reachable.
- If an object on the heap is reachable and contains a pointer to another object in the heap, then the second object is also reachable.
- An object on the heap is **unreachable** if it is not reachable from a live variable, or a heap object that is reachable.

Reachability

The key idea of **reachability** is this, for any point in a program we can work out which variables are **live** as follows:

- We look at the program syntax (known at compile time) to see which variables are in scope. These variables are called **roots**.
- If a root is a pointer to an object on the heap, (e.g. the memory `a` points to in `a = new A(...)`, or in `a = new int [5]`, then that object is reachable.
- If an object on the heap is reachable and contains a pointer to another object in the heap, then the second object is also reachable.
- An object on the heap is **unreachable** if it is not reachable from a live variable, or a heap object that is reachable.

That's it ...

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.
- When you run out, interrupt the program execution and invoke the GC, which does the following.

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.
- When you run out, interrupt the program execution and invoke the GC, which does the following.
  - Get the set of live variables at the point of program interruption (the variables on the stack and in registers).

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.
- When you run out, interrupt the program execution and invoke the GC, which does the following.
  - Get the set of live variables at the point of program interruption (the variables on the stack and in registers).
  - Compute the set of reachable and unreachable heap memory cells.

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.
- When you run out, interrupt the program execution and invoke the GC, which does the following.
  - Get the set of live variables at the point of program interruption (the variables on the stack and in registers).
  - Compute the set of reachable and unreachable heap memory cells.
  - Add all the unreachable cells to the free list.

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.
- When you run out, interrupt the program execution and invoke the GC, which does the following.
  - Get the set of live variables at the point of program interruption (the variables on the stack and in registers).
  - Compute the set of reachable and unreachable heap memory cells.
  - Add all the unreachable cells to the free list.
- Resume program execution.

Garbage collection

With the concept of (un)reachability in mind we can describe automatic memory management as follows:

- Maintain a list of free memory on the heap, initially containing the whole heap.
- Freely allocate memory from the free list whenever `new` is called, until you run out.
- When you run out, interrupt the program execution and invoke the GC, which does the following.
  - Get the set of live variables at the point of program interruption (the variables on the stack and in registers).
  - Compute the set of reachable and unreachable heap memory cells.
  - Add all the unreachable cells to the free list.
- Resume program execution.

Many variants of this scheme exist, e.g. run the GC not when memory has run out, but periodically.

Live variables - Example

Live variables - Example

What are the live variables at program points 1, ..., 5?

```
A a = new A();
               // point 1
while ( true ) {
               // point 2
   NetConnection conn = new NetConnection ( ... );
               // point 3
   Customer cust = new Customer ( conn );
               // point 4
   ...
   if ( ... ) break;
}
               // point 5
```

Solve in class

How can we implement GC at run-time?

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

We must work out what (if any) pointers to the heap our source language has.

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

We must work out what (if any) pointers to the heap our source language has.

We must determine how to find the roots, i.e. the variables that are live **at the point in the program's execution when we run GC**.

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

We must work out what (if any) pointers to the heap our source language has.

We must determine how to find the roots, i.e. the variables that are live **at the point in the program's execution when we run GC**.

We must work out the embedded pointers (if any) of any piece of heap memory.

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

We must work out what (if any) pointers to the heap our source language has.

We must determine how to find the roots, i.e. the variables that are live **at the point in the program's execution when we run GC**.

We must work out the embedded pointers (if any) of any piece of heap memory.

We must decide on an algorithm that finds all (un)reachable memory cells.

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

We must work out what (if any) pointers to the heap our source language has.

We must determine how to find the roots, i.e. the variables that are live **at the point in the program's execution when we run GC**.

We must work out the embedded pointers (if any) of any piece of heap memory.

We must decide on an algorithm that finds all (un)reachable memory cells.

We must then reclaim the unreachable memory.

How can we implement GC at run-time?

This depends on the language we use, and the compilation strategy.

We must work out what (if any) pointers to the heap our source language has.

We must determine how to find the roots, i.e. the variables that are live **at the point in the program's execution when we run GC**.

We must work out the embedded pointers (if any) of any piece of heap memory.

We must decide on an algorithm that finds all (un)reachable memory cells.

We must then reclaim the unreachable memory.

There are many different ways to do this.

What are the pointers in a language?

Consider the simple language we used to explain code generation.

$$
\begin{array}{rcl}
D & \rightarrow & \texttt{def}\ I(A) = E \\
E & \rightarrow & INT \mid ID \mid \texttt{if}\ E = E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & E + E \mid E - E \mid ID(EA)
\end{array}
$$

What are the pointers in a language?

Consider the simple language we used to explain code generation.

$$
\begin{array}{lcl}
D & \rightarrow & \texttt{def}\ I\,(A)\ =\ E \\
E & \rightarrow & INT \mid ID \mid \texttt{if}\ E = E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & E + E \mid E - E \mid ID(EA)
\end{array}
$$

Question: What are the pointers/references into the heap?

What are the pointers in a language?

Consider the simple language we used to explain code generation.

$$\begin{array}{rcl} D & \rightarrow & \texttt{def}\ I(A) = E \\ E & \rightarrow & INT \mid ID \mid \texttt{if}\ E = E\ \texttt{then}\ E\ \texttt{else}\ E \\ & \mid & E + E \mid E - E \mid ID(EA) \end{array}$$

Question: What are the pointers/references into the heap?

Answer: there are none. All variables are procedure parameters. They are stored on the stack in activation records. They live on the stack and are automatically freed when the corresponding AR is popped of the stack.

What are the pointers in a language

In a language like Java, pointers into the heap are created whenever you call `new`. So in

```
A   a = new A ( ... );
B[] b = new B[] { ... };
int n = 3;
```

the variables `a` and `b` **point** into the heap, unlike `n`.

What are the pointers in a language

In a language like Java, pointers into the heap are created whenever you call `new`. So in

```
A   a = new A ( ... );
B[] b = new B[] { ... };
int n = 3;
```

the variables `a` and `b` **point** into the heap, unlike `n`.

But GC executes at run-time, so to find the roots at run-time, we must know where variables are located.

What are the pointers in a language

Remembering our (accumulator machine) compilation, variables are stored in:

- The accumulator. Other compilation schemes use more registers.
- Activation records on the stack.

What are the pointers in a language

Remembering our (accumulator machine) compilation, variables are stored in:

- The accumulator. Other compilation schemes use more registers.
- Activation records on the stack.

Question: How does the GC know at run-time if a register or an activation record entry contain a pointer into the heap, or an integer that is not a pointer?

What are the pointers in a language

Remembering our (accumulator machine) compilation, variables are stored in:

- The accumulator. Other compilation schemes use more registers.
- Activation records on the stack.

Question: How does the GC know at run-time if a register or an activation record entry contain a pointer into the heap, or an integer that is not a pointer?

Answer: combintion of using type-information available at compile-time, together with type information stored in headers at run-time. Details of this are quite tricky ... (and beyond scope of this course)

What are the pointers in a language

Remember a piece of heap memory `m` is reachable, exactly when one of the following conditions holds:

What are the pointers in a language

Remember a piece of heap memory `m` is reachable, exactly when one of the following conditions holds:

- Another piece of reachable heap memory contains a pointer to `m`.

What are the pointers in a language

Remember a piece of heap memory **`m`** is reachable, exactly when one of the following conditions holds:

- Another piece of reachable heap memory contains a pointer to **`m`**.
- A root (register or activation record) contains a pointer to **`m`** and the root is of a type that is stored on the heap (e.g. not integers). Why is the qualification important?

What are the pointers in a language

Remember a piece of heap memory `m` is reachable, exactly when one of the following conditions holds:

- Another piece of reachable heap memory contains a pointer to `m`.
- A root (register or activation record) contains a pointer to `m` and the root is of a type that is stored on the heap (e.g. not integers). Why is the qualification important?
  - Integers might happen to have values that match the addresses of objects on the heap

What are the pointers in a language

Remember a piece of heap memory `m` is reachable, exactly when one of the following conditions holds:

- Another piece of reachable heap memory contains a pointer to `m`.
- A root (register or activation record) contains a pointer to `m` and the root is of a type that is stored on the heap (e.g. not integers). Why is the qualification important?
  - Integers might happen to have values that match the addresses of objects on the heap

Languages where integers can be used as pointers (i.e. C/C++) cannot really be GCed (at least not well) for this reason.

Reachability is an approximation

Reachability is an approximation

Consider the following slightly artificial program.

```
A tmp = new A ();
   // can mem pointed to by "a"
   // affect future of computation?
if ( ... ) {
   B b = new B() // might trigger GC
                 // tmp is root here
   b.f( 17 )
   return "hello" }
else { ... }
... // tmp is used here
```

Reachability is an approximation

Consider the following slightly artificial program.

```
A tmp = new A ();
   // can mem pointed to by "a"
   // affect future of computation?
if ( ... ) {
   B b = new B() // might trigger GC
                 // tmp is root here
   b.f( 17 )
   return "hello" }
else { ... }
... // tmp is used here
```

After the first line, the memory allocated by `new` in the first line can never be used again, hence cannot affect rest of computation. But it is reachable (from root `tmp` which is in scope in `then ...`), so cannot be GCed inside of `then`.

Reachability must be an approximation

Programs like

```
A tmp = new A ();
if ( ... ) {
   B b = new B() // might trigger GC
                 // tmp is root here
...
```

show that reachability at a given program point is an **approximation**: a piece of memory might be reachable, but cannot be used by the program.

Reachability must be an approximation

Programs like

```
A tmp = new A ();
if ( ... ) {
   B b = new B() // might trigger GC
                 // tmp is root here
...
```

show that reachability at a given program point is an **approximation**: a piece of memory might be reachable, but cannot be used by the program.

Cause: roots (which start reachability search) is a compile-time concept, and we assume that all roots are reachable.

Reachability must be an approximation

Programs like

```
A tmp = new A ();
if ( ... ) {
   B b = new B() // might trigger GC
                 // tmp is root here
...
```

show that reachability at a given program point is an **approximation**: a piece of memory might be reachable, but cannot be used by the program.

Cause: roots (which start reachability search) is a compile-time concept, and we assume that all roots are reachable.

Approximations are necessary! To be on the safe side, the memory allocated by the initial `new` is considered reachable before and inside the loop.

(Remember Rice's theorem?)

The algorithm for computing reachability

```
reach( roots ) =
  for each r in roots
    if ( r not yet processed )
      mark heap memory that r points to as reachable
      let [r1, ..., rn] be the pointers contained in r
      reach( [r1, ..., rn] )
```

The algorithm for computing reachability

```
reach( roots ) =
  for each r in roots
    if ( r not yet processed )
      mark heap memory that r points to as reachable
      let [r1, ..., rn] be the pointers contained in r
      reach( [r1, ..., rn] )
```

How do we mark memory as reachable?

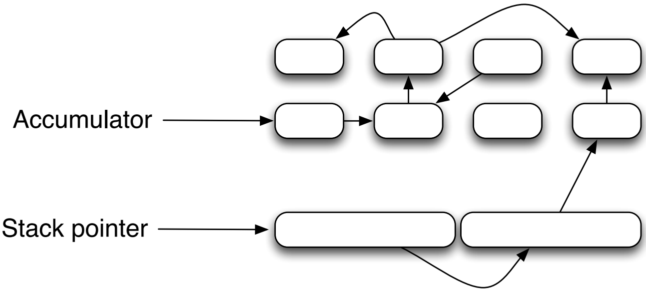
The algorithm for computing reachability

```
reach( roots ) =
  for each r in roots
    if ( r not yet processed )
      mark heap memory that r points to as reachable
      let [r1, ..., rn] be the pointers contained in r
      reach( [r1, ..., rn] )
```

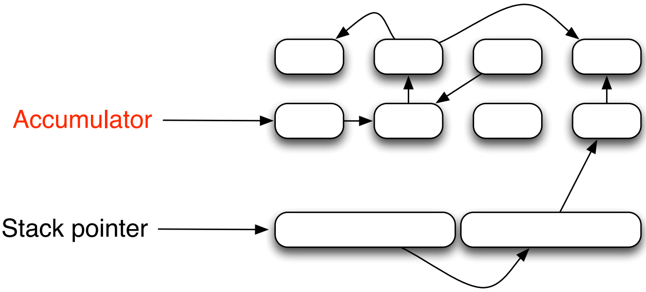
How do we mark memory as reachable?

There are several ways of mark memory as reachable. In OO languages, we reserve a bit in the header of each object that can be stored in the heap.

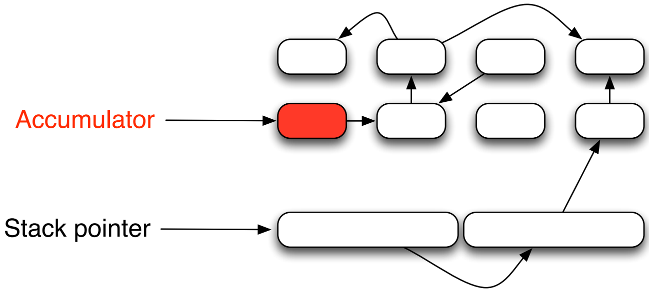
The algorithm for computing reachability



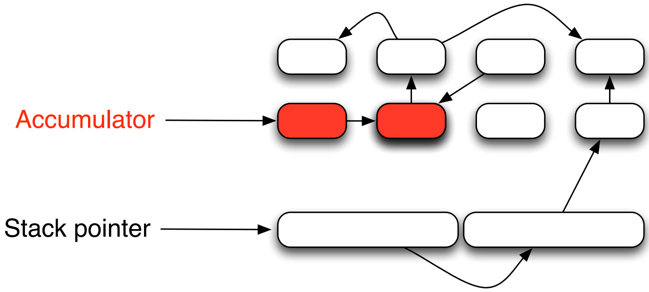
The algorithm for computing reachability



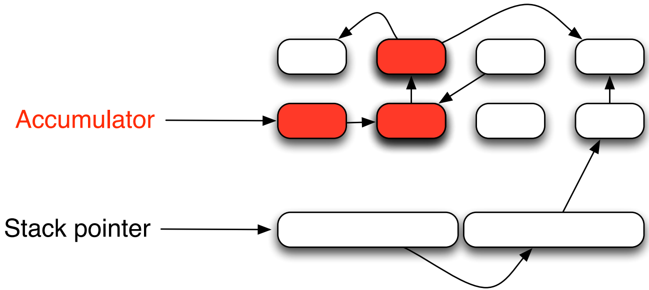
The algorithm for computing reachability



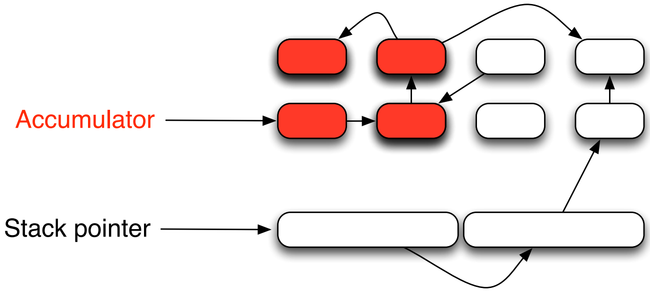
The algorithm for computing reachability



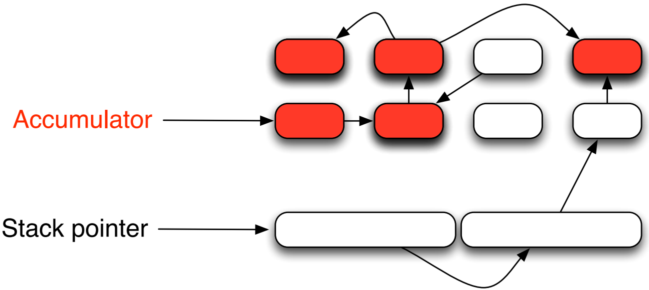
The algorithm for computing reachability



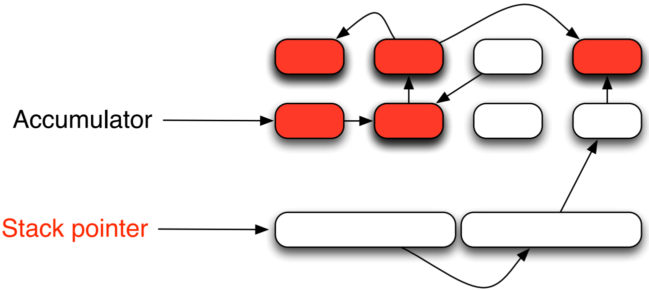
The algorithm for computing reachability



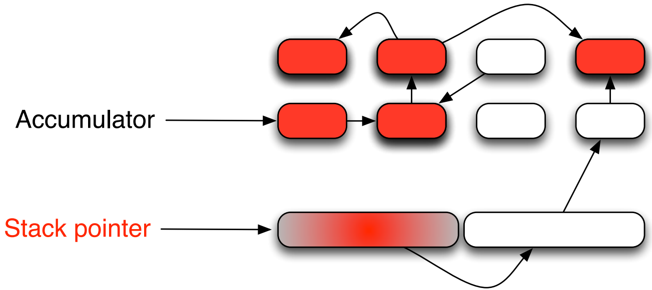
The algorithm for computing reachability



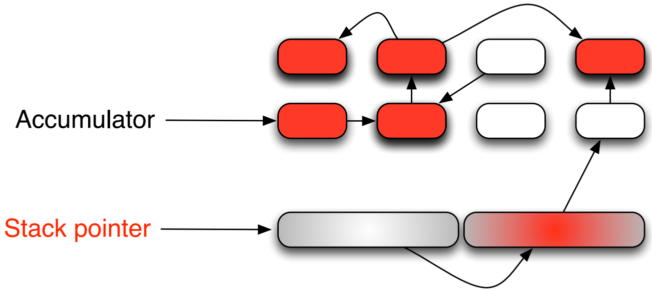
The algorithm for computing reachability



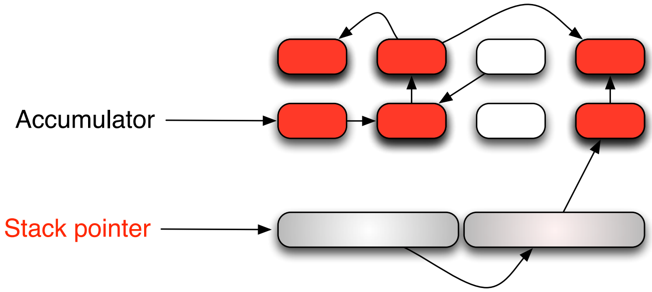
The algorithm for computing reachability



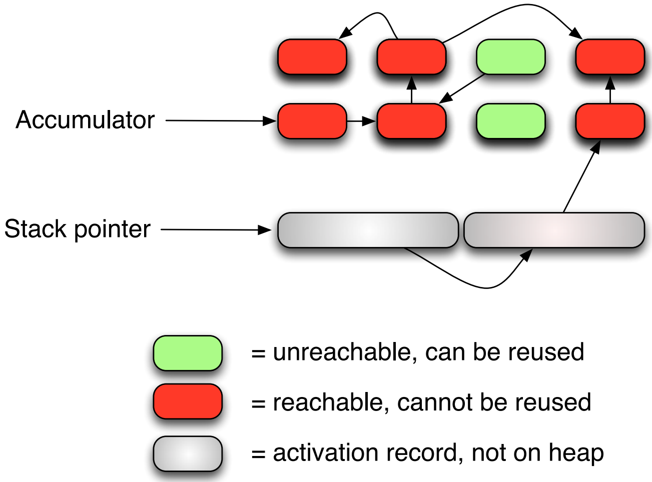
The algorithm for computing reachability



The algorithm for computing reachability



The algorithm for computing reachability



Two GC algorithms

We are going to look at two GC algorithms.

- Mark and sweep
- Stop and copy

Mark and sweep

The mark and sweep algorithm has two phases:

Mark and sweep

The mark and sweep algorithm has two phases:

- The **mark phase** which computes the reachable parts of the heap, starting from the roots.

Mark and sweep

The mark and sweep algorithm has two phases:

- The **mark phase** which computes the reachable parts of the heap, starting from the roots.
- The **sweep phase** which reclaims unreachable memory, by going through whole heap.

Mark and sweep

The mark and sweep algorithm has two phases:

- The **mark phase** which computes the reachable parts of the heap, starting from the roots.
- The **sweep phase** which reclaims unreachable memory, by going through whole heap.

For this to work (easily) we need need the following information.

Mark and sweep

The mark and sweep algorithm has two phases:

- The **mark phase** which computes the reachable parts of the heap, starting from the roots.
- The **sweep phase** which reclaims unreachable memory, by going through whole heap.

For this to work (easily) we need need the following information.

- Mark bit, initially set to 0 when memory is allocated.

Mark and sweep

The mark and sweep algorithm has two phases:

- The **mark phase** which computes the reachable parts of the heap, starting from the roots.
- The **sweep phase** which reclaims unreachable memory, by going through whole heap.

For this to work (easily) we need need the following information.

- Mark bit, initially set to 0 when memory is allocated.
- Size of the object in memory, and location of pointers embedded in an object. This can often be determined using types.

Mark and sweep

The mark and sweep algorithm has two phases:

- The **mark phase** which computes the reachable parts of the heap, starting from the roots.
- The **sweep phase** which reclaims unreachable memory, by going through whole heap.

For this to work (easily) we need need the following information.

- Mark bit, initially set to 0 when memory is allocated.
- Size of the object in memory, and location of pointers embedded in an object. This can often be determined using types.

In addition, we need a **free list** which holds the memory that is available for allocation (free).

Mark phase

We run the algorithm shown a few slides ago that sets the mark bit to 1 in each heap object reachable from the roots.

```
reach( roots ) =
  for each r in roots
    if ( r not yet processed )
      r.markbit := 1
      let [r1, ..., rn] be the pointers contained in r
      reach( [r1, ..., rn] )
```

Sweep phase

Conceptually, this phase proceeds in two steps.

Sweep phase

Conceptually, this phase proceeds in two steps.

- The algorithm scans the heap looking for objects with mark bit zero. Any such object is added to the free list. We know the size of the memory part we add to the free list because the size is in the header.

Sweep phase

Conceptually, this phase proceeds in two steps.

- The algorithm scans the heap looking for objects with mark bit zero. Any such object is added to the free list. We know the size of the memory part we add to the free list because the size is in the header.
- We reset the mark bit to 0 in objects where it is 1.

Sweep phase

Conceptually, this phase proceeds in two steps.

- The algorithm scans the heap looking for objects with mark bit zero. Any such object is added to the free list. We know the size of the memory part we add to the free list because the size is in the header.
- We reset the mark bit to 0 in objects where it is 1.

```
sweep( heap ) =
   for each cell c in heap
      if ( c.markbit == 1 )
         c.markbit := 0
      else
         putOnFreeList ( c )
```

Mark and sweep

Conceptually, this algorithm is really simple. But it has a number of really tricky details (typical for GC algorithms).

Mark and sweep

Conceptually, this algorithm is really simple. But it has a number of really tricky details (typical for GC algorithms).

A problem with the mark phase:

Mark and sweep

Conceptually, this algorithm is really simple. But it has a number of really tricky details (typical for GC algorithms).

A problem with the mark phase:

The GC is invoked when we run out of memory.

Mark and sweep

Conceptually, this algorithm is really simple. But it has a number of really tricky details (typical for GC algorithms).

A problem with the mark phase:

The GC is invoked when we run out of memory.

But we need space to run the marking algorithm, and later for resetting the mark bits to 0 (at least when we implement this naively). If this space is not known at compile time, how can we reserve space for it?

Mark and sweep

Conceptually, this algorithm is really simple. But it has a number of really tricky details (typical for GC algorithms).

A problem with the mark phase:

The GC is invoked when we run out of memory.

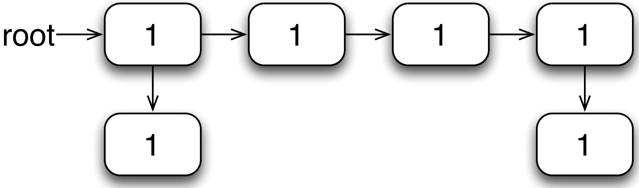
But we need space to run the marking algorithm, and later for resetting the mark bits to 0 (at least when we implement this naively). If this space is not known at compile time, how can we reserve space for it?

There are some very neat algorithms for this purpose (google Schorr-Waite graph marking algorithm aka Pointer Reversal).

Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

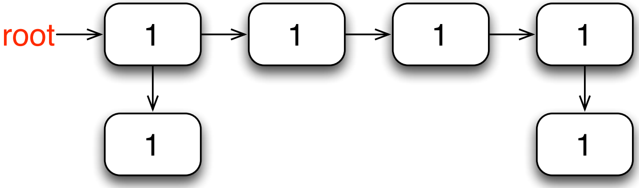
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

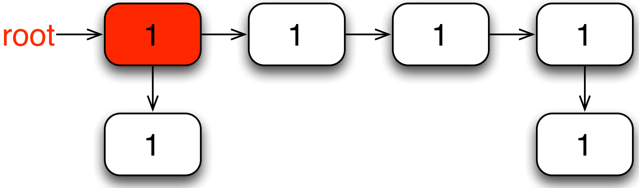
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

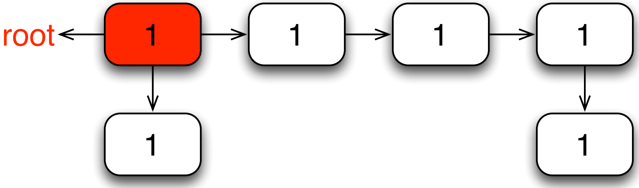
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

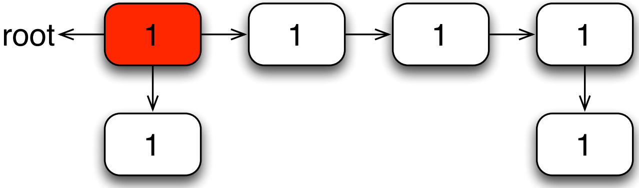
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

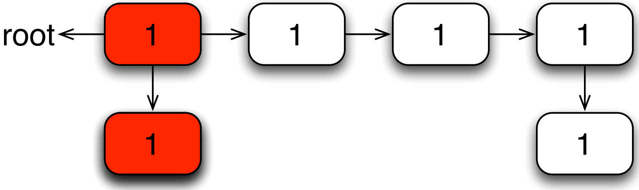
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

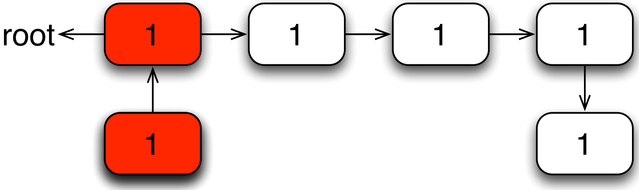
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

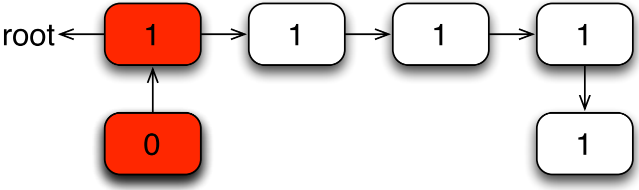
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

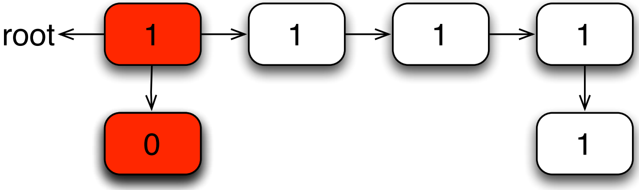
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

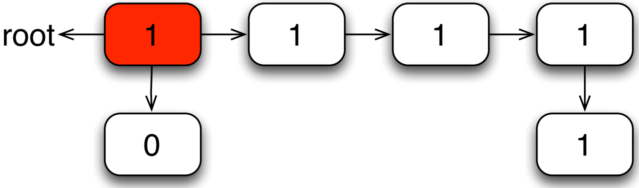
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

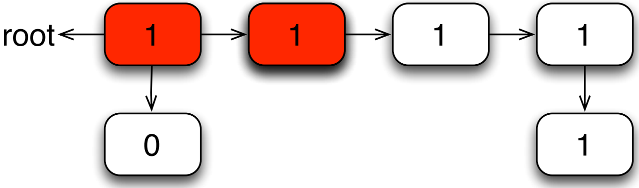
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

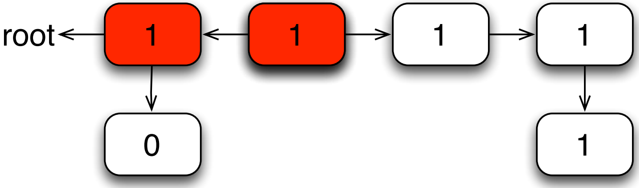
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

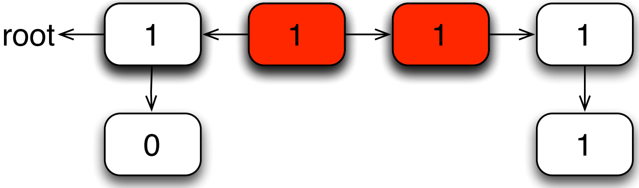
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

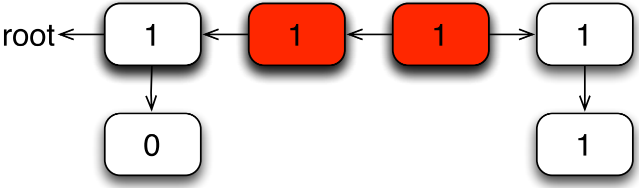
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

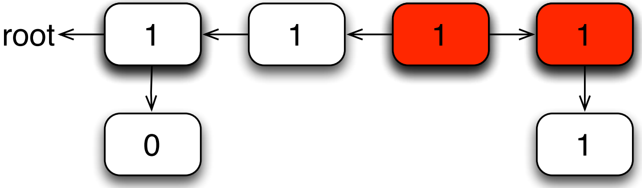
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

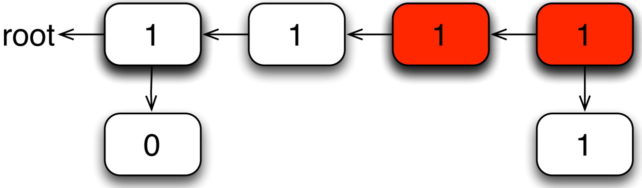
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

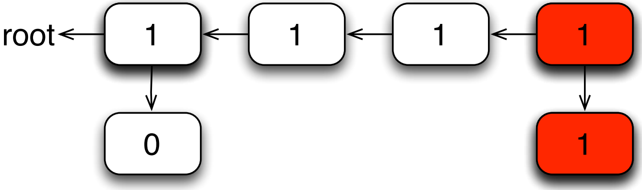
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

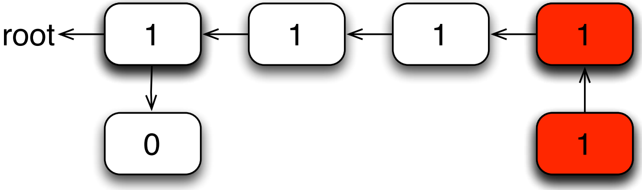
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

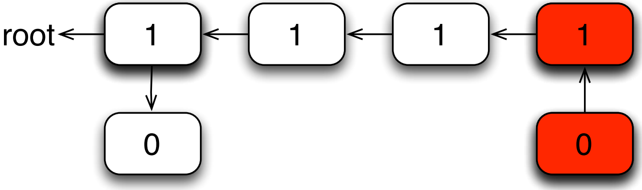
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

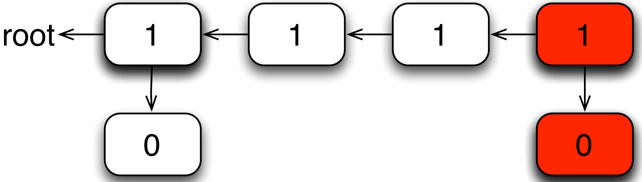
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

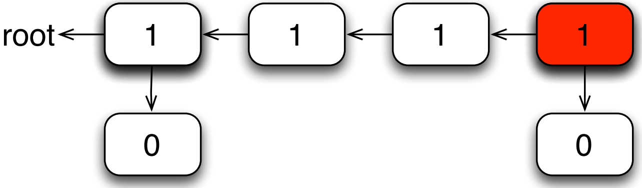
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

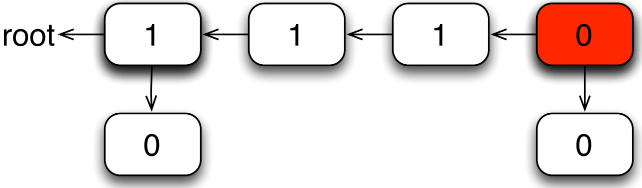
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

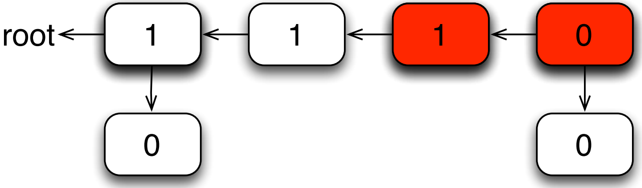
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

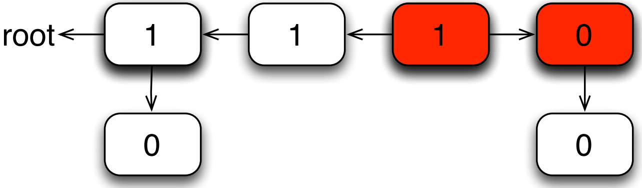
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

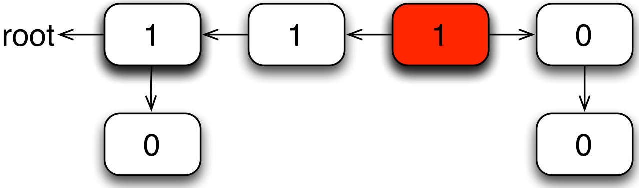
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

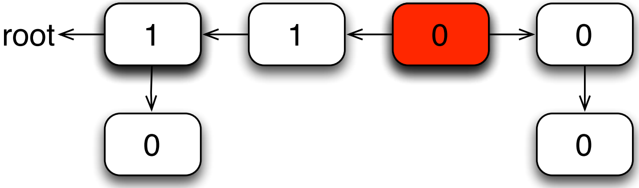
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

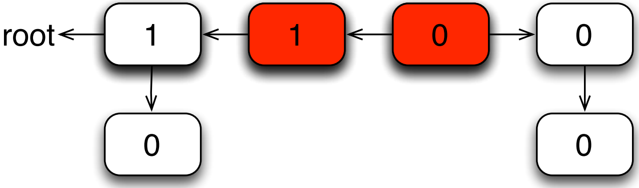
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

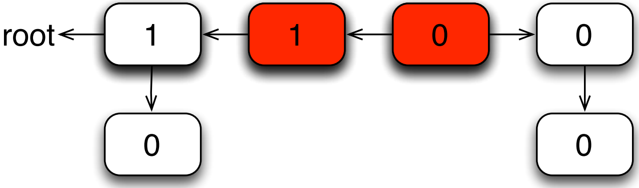
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

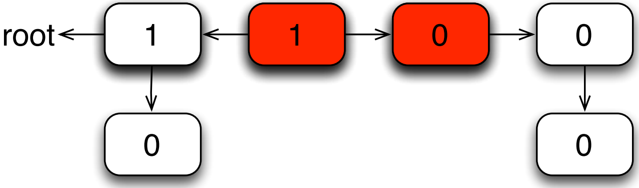
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

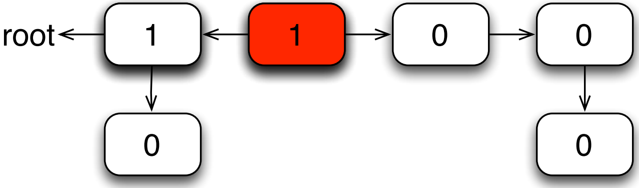
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

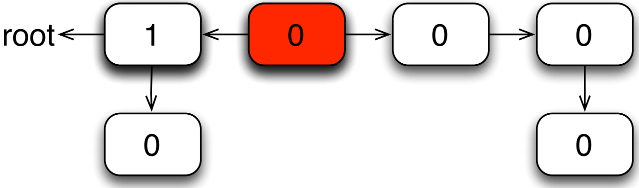
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

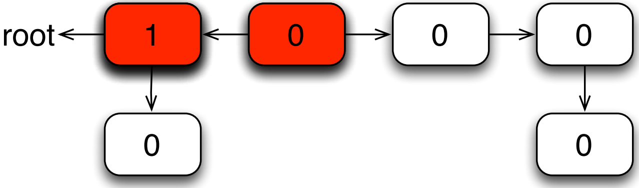
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

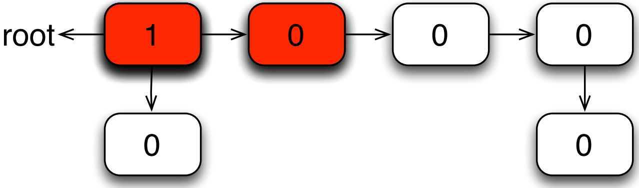
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

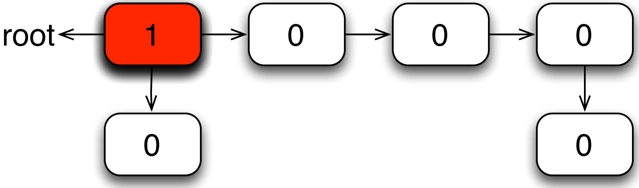
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

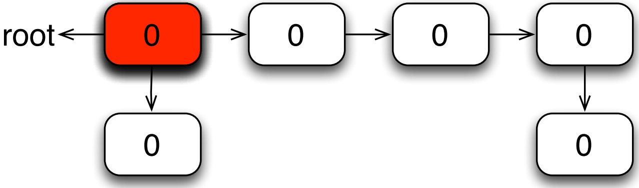
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

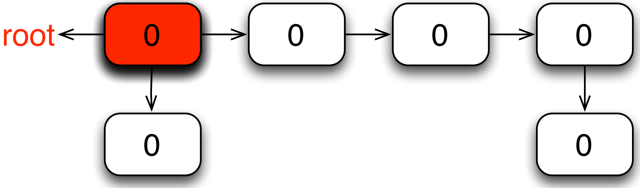
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

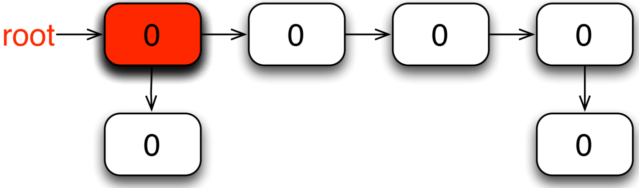
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

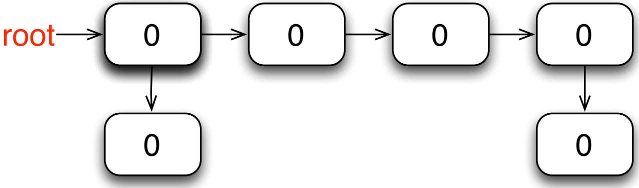
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

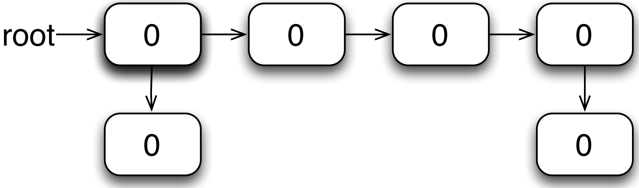
- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Schorr-Waite algorithm (not exam relevant)

A clever algorithm to explore a graph without additional memory (except one pointer). It can be used to reset the mark bits. The key idea is:

- Follow a pointer to a new node
- Reverse the pointer we just followed so we can go back
- Explore the pointers at the new location
- When we go back, we reverse the pointer again.



Constructing the free list

The Schorr-Waite algorithm enables us to sweep the reachable objects (essentially) without using additional memory.

Constructing the free list

The Schorr-Waite algorithm enables us to sweep the reachable objects (essentially) without using additional memory.

But how do we construct the new free list without using additional memory?

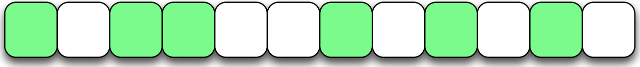
Constructing the free list

The Schorr-Waite algorithm enables us to sweep the reachable objects (essentially) without using additional memory.

But how do we construct the new free list without using additional memory?

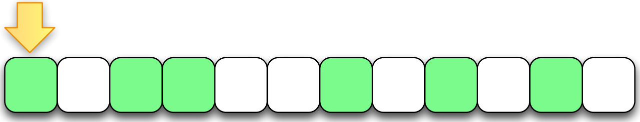
Idea: use the free space itself to construct the free list.

Constructing the free list



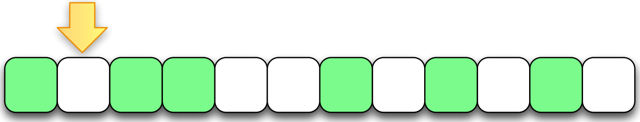
free

Constructing the free list



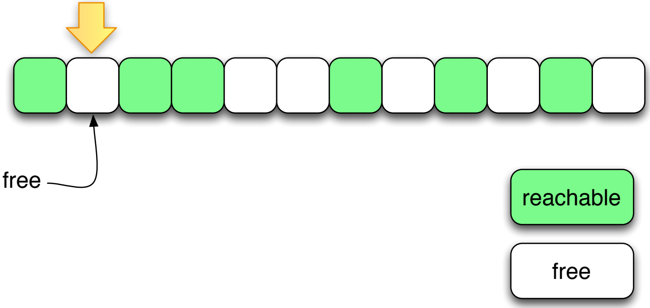
free

Constructing the free list

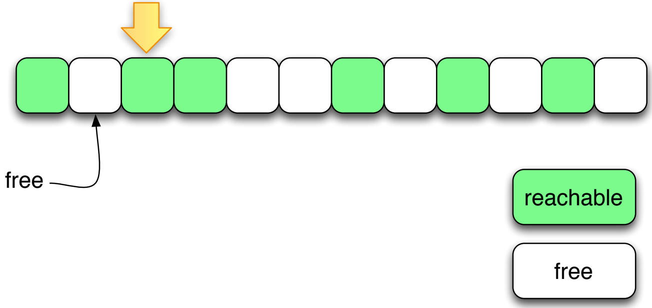


free

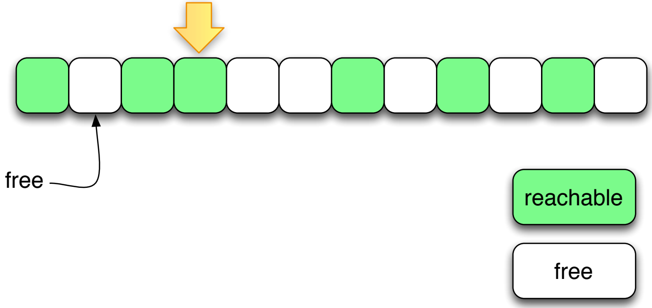
Constructing the free list



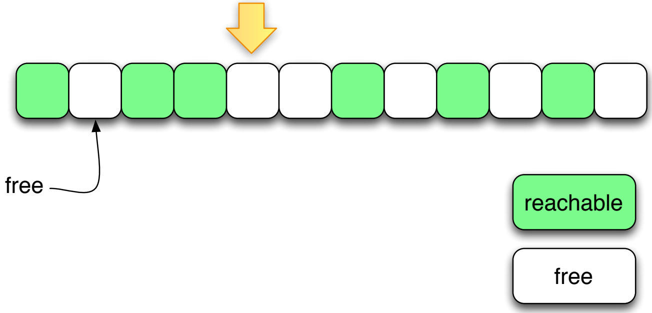
Constructing the free list



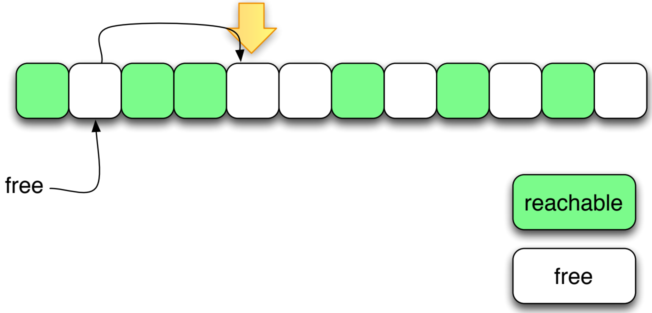
Constructing the free list



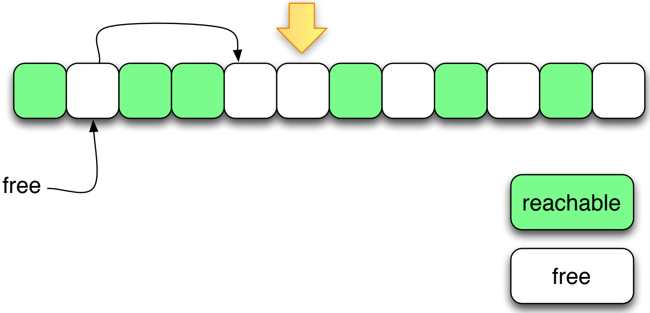
Constructing the free list



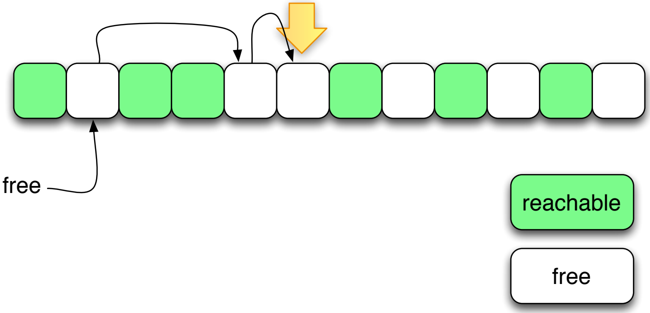
Constructing the free list



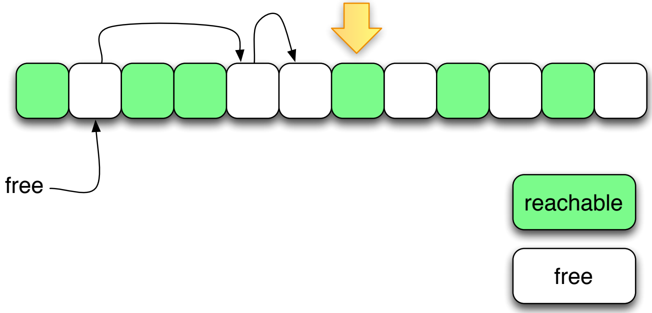
Constructing the free list



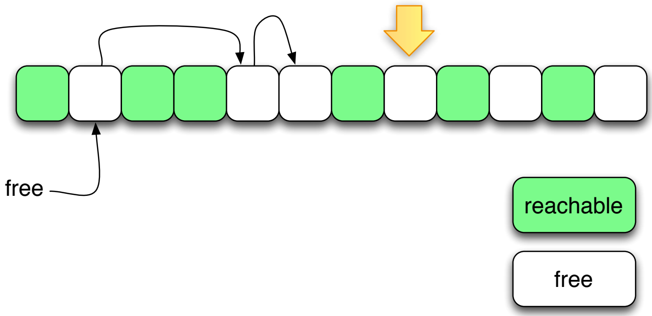
Constructing the free list



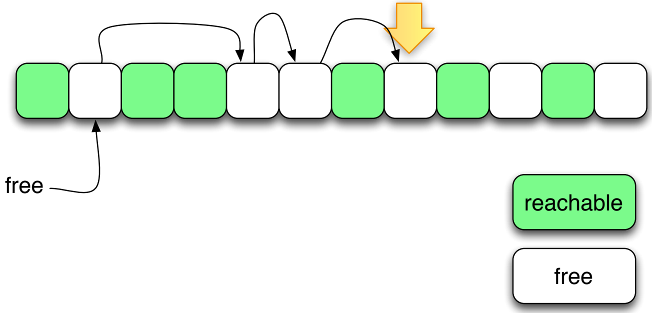
Constructing the free list



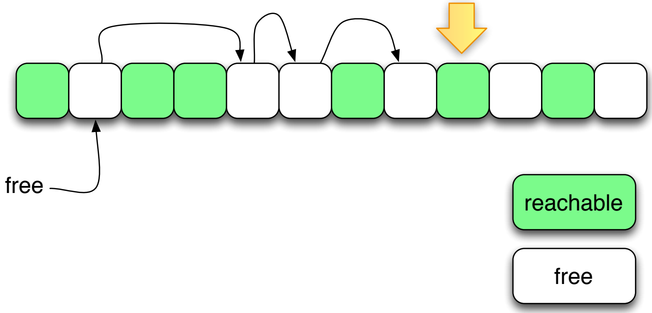
Constructing the free list



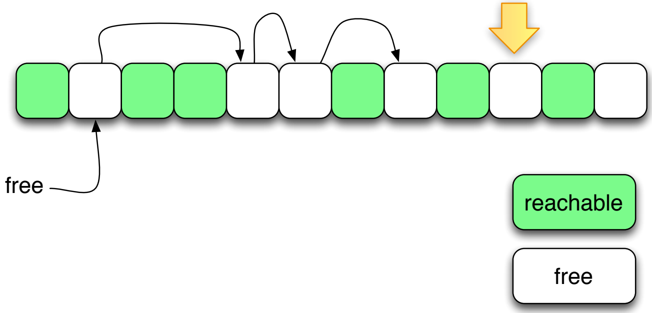
Constructing the free list



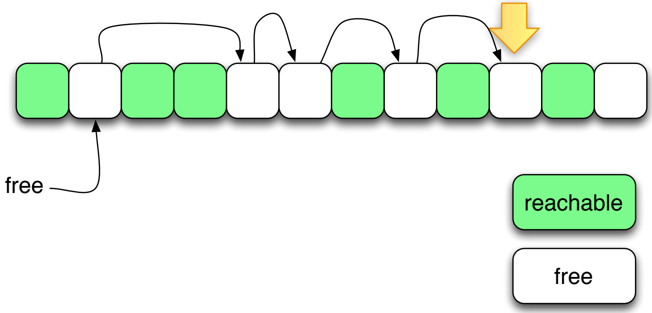
Constructing the free list



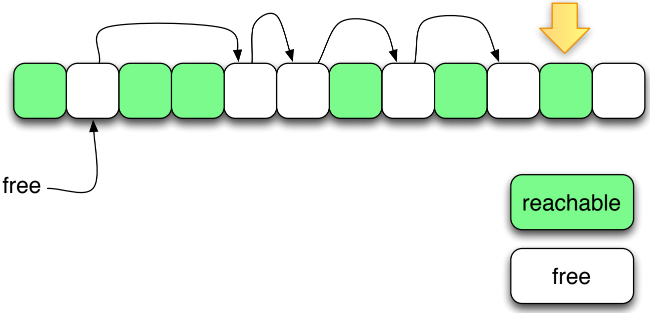
Constructing the free list



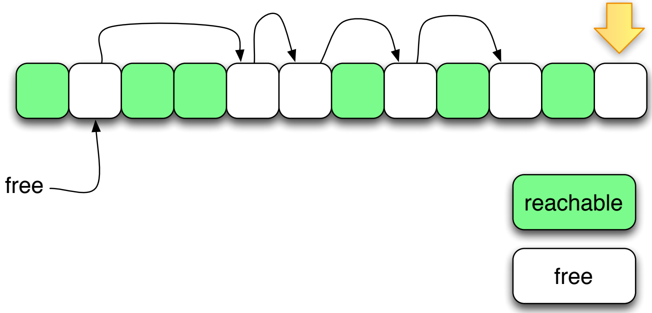
Constructing the free list



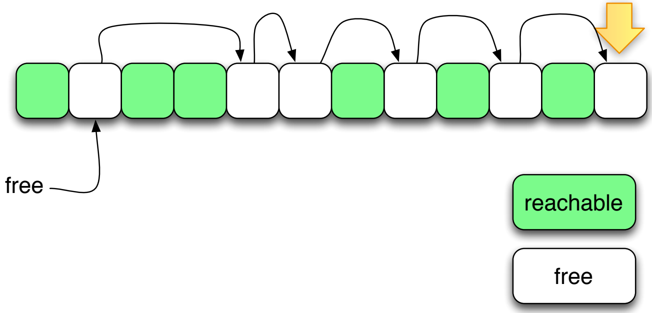
Constructing the free list



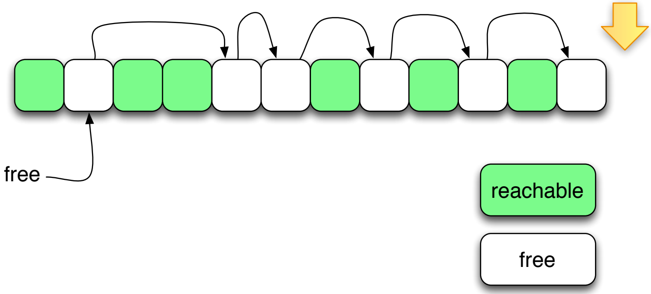
Constructing the free list



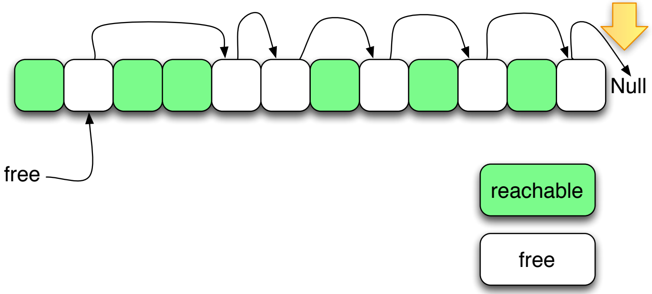
Constructing the free list



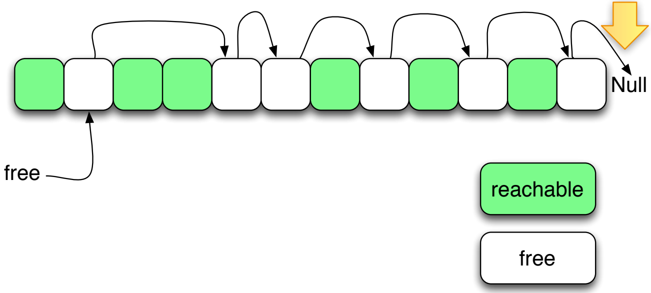
Constructing the free list



Constructing the free list

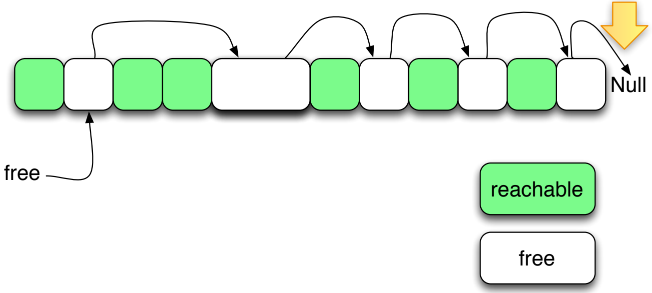


Constructing the free list



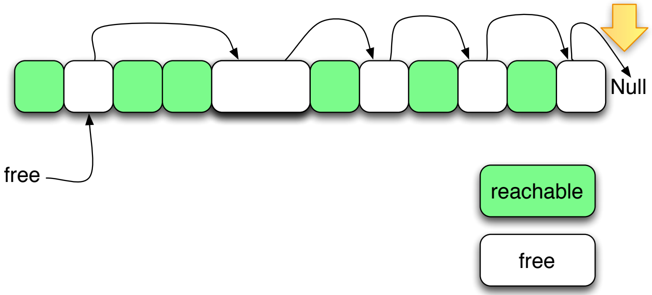
In practise we also merge adjacent free memory regions into larger regions, so we can later use them to allocate bigger objects.

Constructing the free list



In practise we also merge adjacent free memory regions into larger regions, so we can later use them to allocate bigger objects.

Constructing the free list



In practise we also merge adjacent free memory regions into larger regions, so we can later use them to allocate bigger objects.

Clearly this way of constructing the free list also doesn't need (much) additional memory, so we can use it if we run out of heap space.

Stop & Copy

One disadvantage of Mark & Sweep is that it has to go throught the **whole** memory **twice** in the worst case, first to mark, then to reset the mark bits and to update the free list. This is expensive, because memory access is slow.

Stop & Copy

One disadvantage of Mark & Sweep is that it has to go throught the **whole** memory **twice** in the worst case, first to mark, then to reset the mark bits and to update the free list. This is expensive, because memory access is slow.

A different GC technique is Stop & Copy. It needs to touch only reachable memory. (When is that especially advantageous?)

Stop & Copy

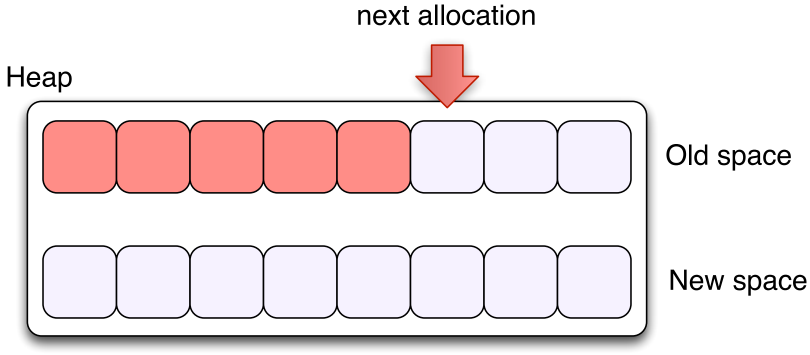
One disadvantage of Mark & Sweep is that it has to go throught the **whole** memory **twice** in the worst case, first to mark, then to reset the mark bits and to update the free list. This is expensive, because memory access is slow.

A different GC technique is Stop & Copy. It needs to touch only reachable memory. (When is that especially advantageous?)

Here is how Stop & Copy works.

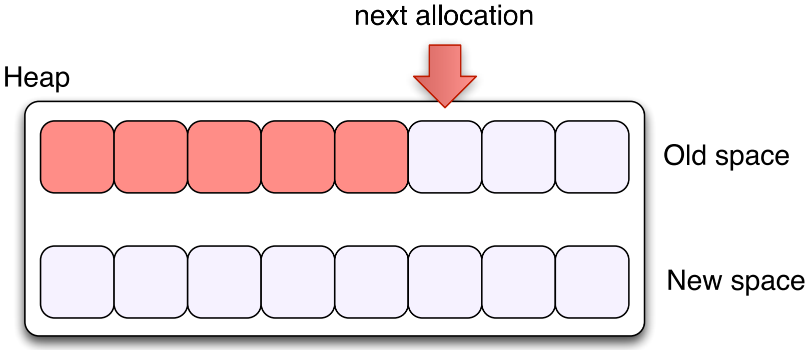
Stop & Copy

Heap is split into two halves:



Stop & Copy

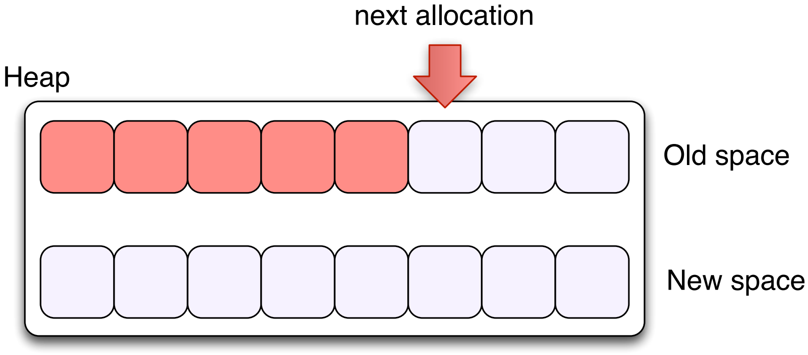
Heap is split into two halves:



Only the Old space is used in the current round. Allocation is simple: just use next free memory at allocation pointer.

Stop & Copy

Heap is split into two halves:

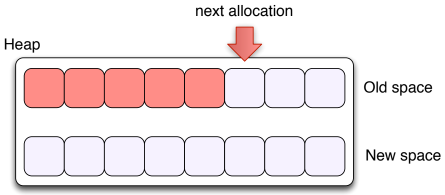


Only the Old space is used in the current round. Allocation is simple: just use next free memory at allocation pointer.

GC happens when the Old space is full.

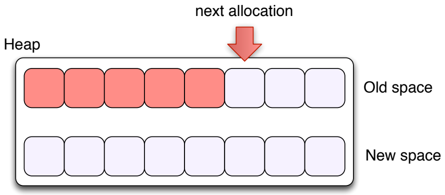
Stop & Copy

Memory is split into two areas (often about equally sized):



Stop & Copy

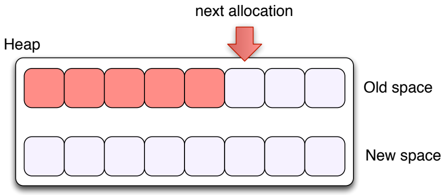
Memory is split into two areas (often about equally sized):



GC finds reachable objects (starting from roots) and **copies** them (instead of marking) from Old space to New space.

Stop & Copy

Memory is split into two areas (often about equally sized):

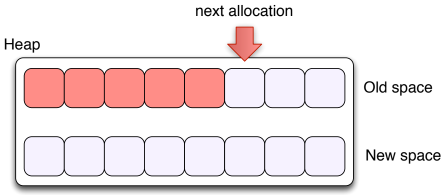


GC finds reachable objects (starting from roots) and **copies** them (instead of marking) from Old space to New space.

Objects contain pointers to other objects. These pointers must be **rewritten** to account for the new location (including roots).

Stop & Copy

Memory is split into two areas (often about equally sized):



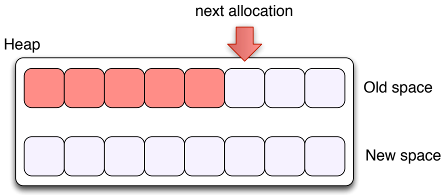
GC finds reachable objects (starting from roots) and **copies** them (instead of marking) from Old space to New space.

Objects contain pointers to other objects. These pointers must be **rewritten** to account for the new location (including roots).

As we copy an object from Old to New, we store a **forwarding** pointer in the old object, pointing to the new version of the object. If we reach an object with a forwarding pointer, we know it was already copied. Like the mark bit, it prevents GC loops.

Stop & Copy

Memory is split into two areas (often about equally sized):



GC finds reachable objects (starting from roots) and **copies** them (instead of marking) from Old space to New space.

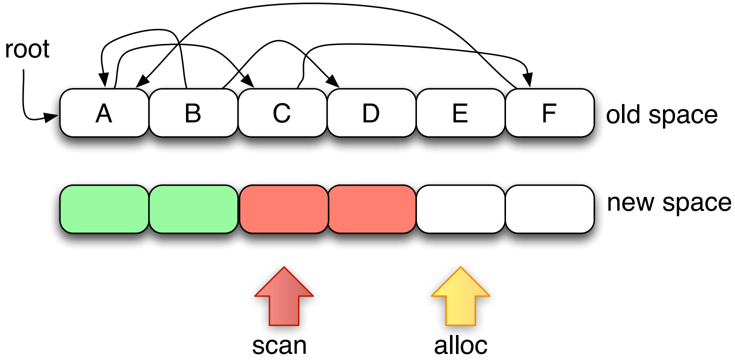
Objects contain pointers to other objects. These pointers must be **rewritten** to account for the new location (including roots).

As we copy an object from Old to New, we store a **forwarding** pointer in the old object, pointing to the new version of the object. If we reach an object with a forwarding pointer, we know it was already copied. Like the mark bit, it prevents GC loops.

Finally we swap the role of Old and New space.

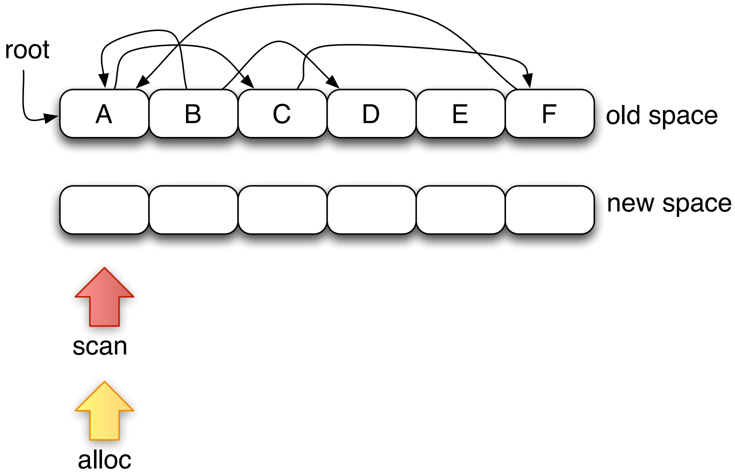
Stop & Copy

To implement the rewriting of addresses after copying, we partition the New space into three parts.

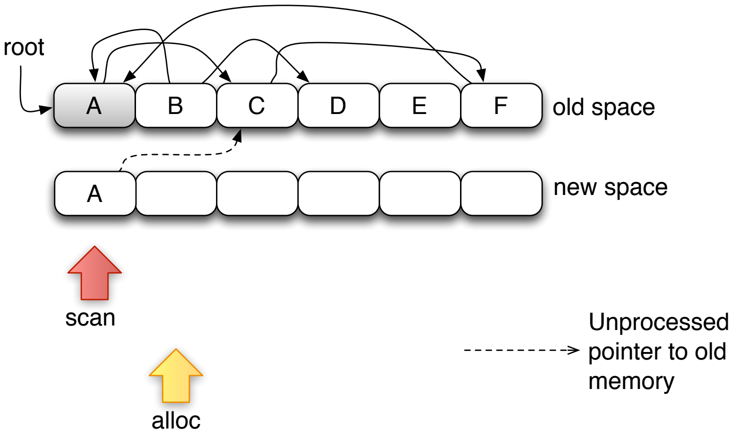


Left of the `scan` pointer, objects are processed (all pointers have been rewritten). The memory between `scan` and `alloc` is copied, but some pointers are still poining to the Old space. Memory from `alloc` on is free.

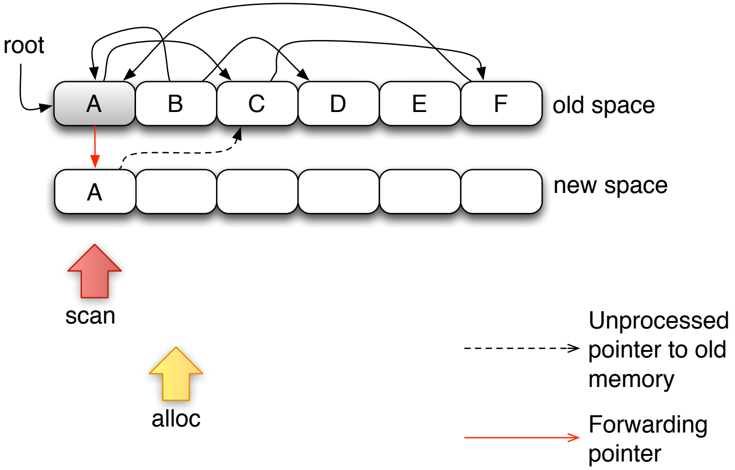
Example of Stop & Copy



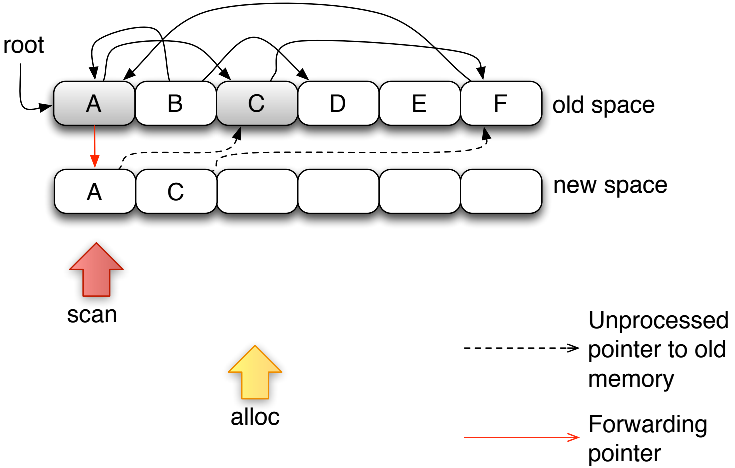
Example of Stop & Copy



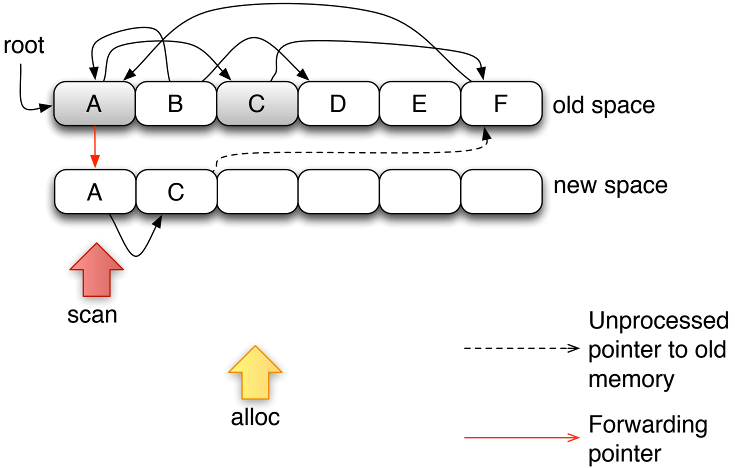
Example of Stop & Copy



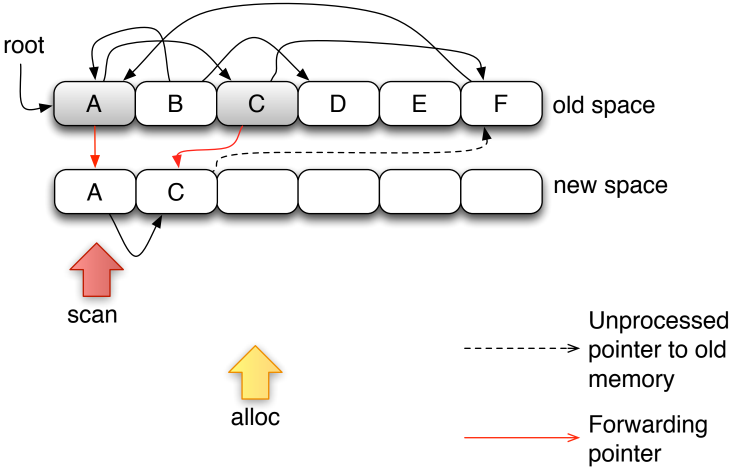
Example of Stop & Copy



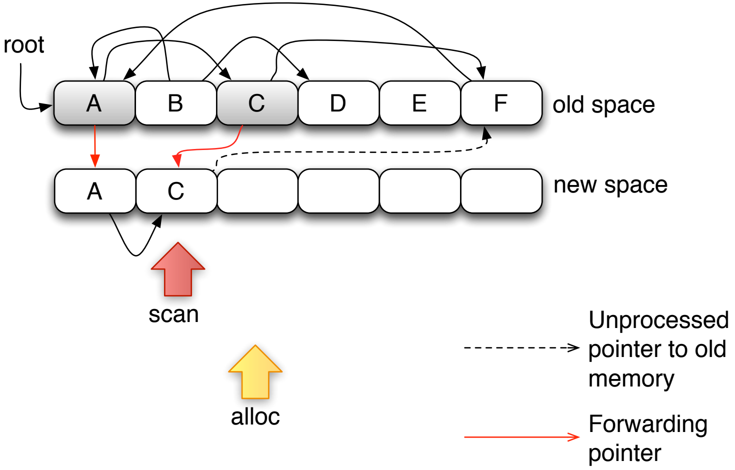
Example of Stop & Copy



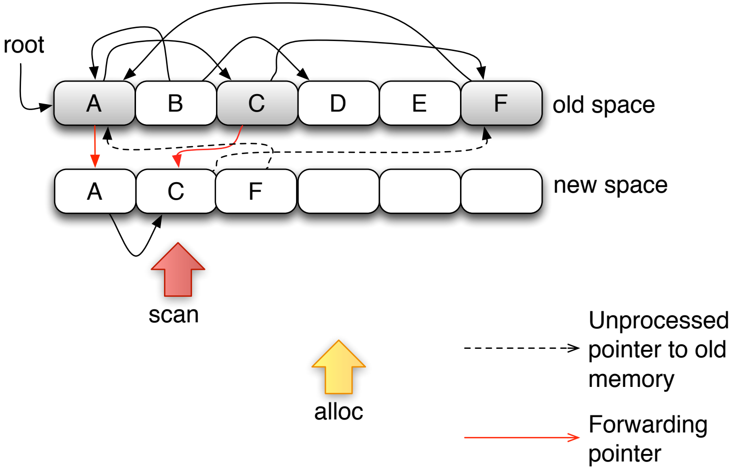
Example of Stop & Copy



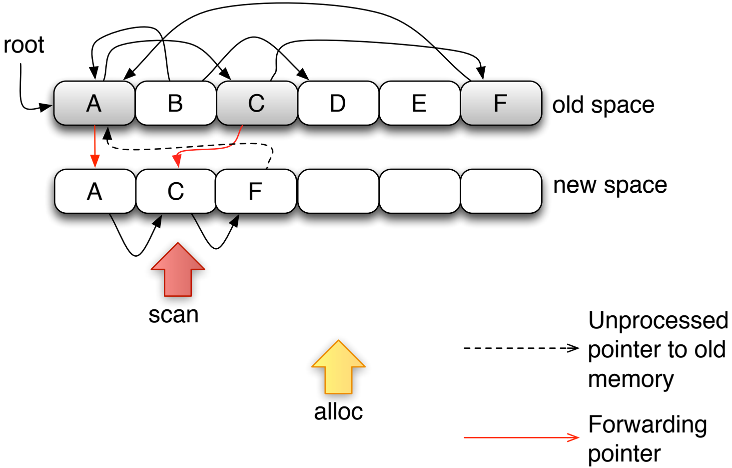
Example of Stop & Copy



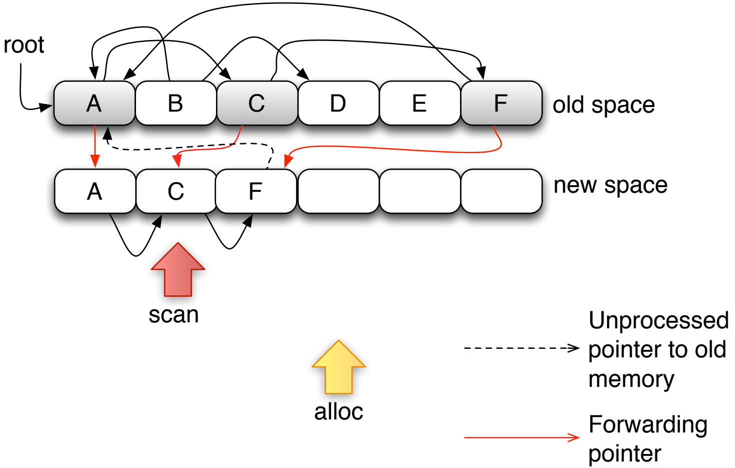
Example of Stop & Copy



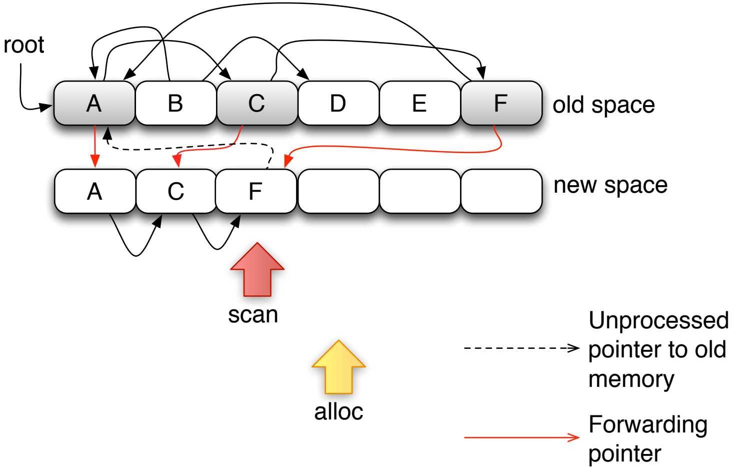
Example of Stop & Copy



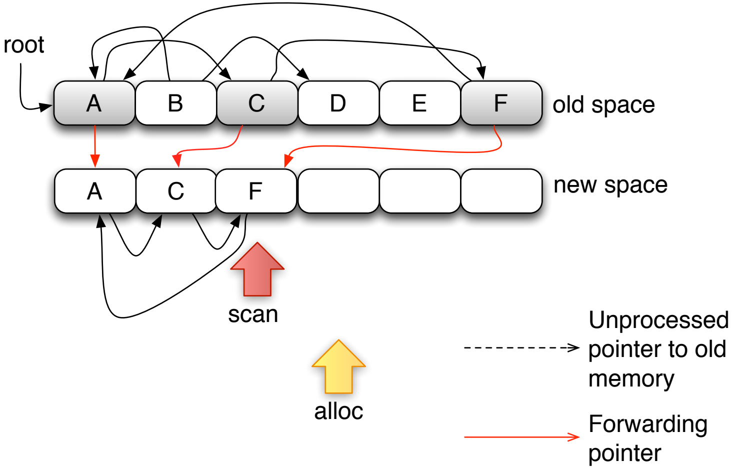
Example of Stop & Copy



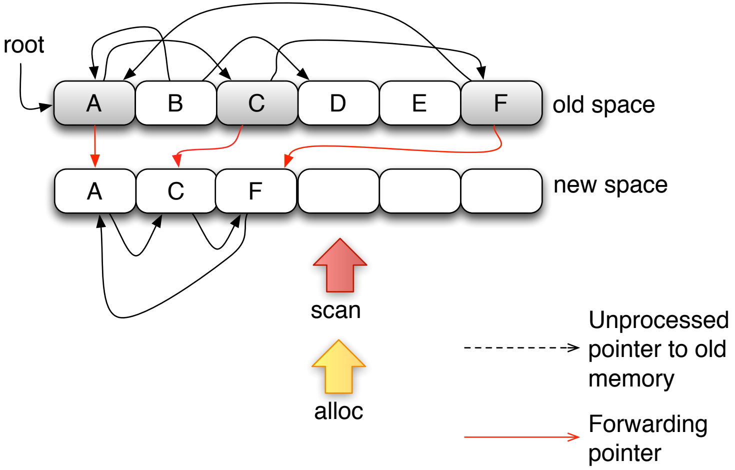
Example of Stop & Copy



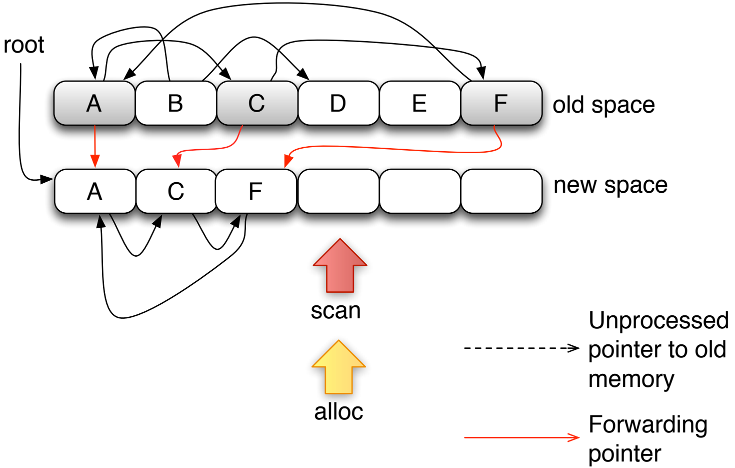
Example of Stop & Copy



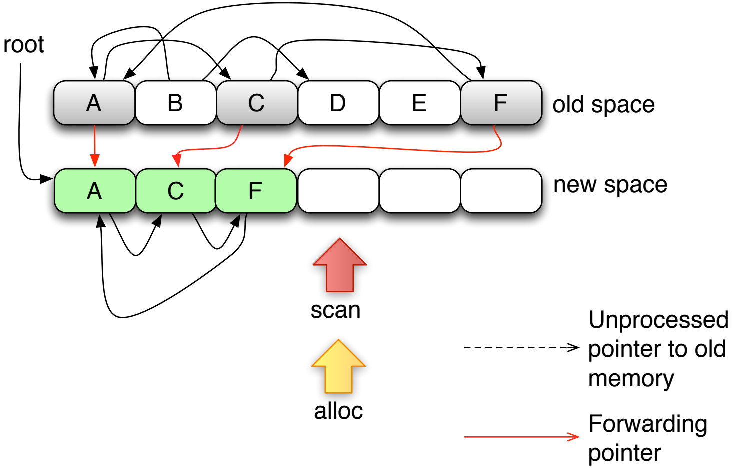
Example of Stop & Copy



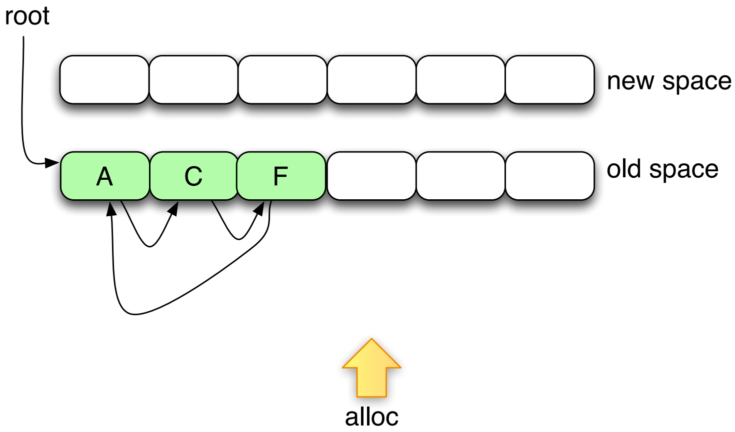
Example of Stop & Copy



Example of Stop & Copy



Example of Stop & Copy



Stop & Copy algorithm

```
while scan < alloc
   let o be the object at scan
   for each embedded pointer p in o
      find o′ that p points to
      if o′ has no forwarding pointer
         copy o′ to new space
         alloc = alloc + size( o′ )
         set forwarding pointer in old o′ to new o′
                      (this marks old o′ as copied)
         change pointer p to point to new o′
      else
         set p in o equal to forwarding pointer
   scan = scan + size( o )
```

We also must rewrite pointers in roots (e.g. activation records).

Stop & Copy

Question: How do we find the pointers embedded in objects?
After all, bit patterns are bit patterns.

Stop & Copy

Question: How do we find the pointers embedded in objects? After all, bit patterns are bit patterns.

Just like with Mark & Sweep, we need to know the size and the types of objects. This can be done by embedding this information in the object's header, and/or using type information. If this information is not available we cannot do GC.

Stop & Copy

Stop & Copy is (in its advanced forms) the fastest GC algorithm.

Stop & Copy

Stop & Copy is (in its advanced forms) the fastest GC algorithm.

Allocation is very fast and simple: just increment the `alloc` pointer. No free list needed.

Stop & Copy

Stop & Copy is (in its advanced forms) the fastest GC algorithm.

Allocation is very fast and simple: just increment the `alloc` pointer. No free list needed.

Collection is relatively cheap, especially if there is a lot of garbage. This is because **only reachable objects are touched**. Does not touch garbage. (Mark & Sweep touches every object in memory.)

Stop & Copy

Stop & Copy is (in its advanced forms) the fastest GC algorithm.

Allocation is very fast and simple: just increment the `alloc` pointer. No free list needed.

Collection is relatively cheap, especially if there is a lot of garbage. This is because **only reachable objects are touched**. Does not touch garbage. (Mark & Sweep touches every object in memory.)

Stop & Copy also improves **memory locality**, which can increase program execution speed.

Stop & Copy

Stop & Copy is (in its advanced forms) the fastest GC algorithm.

Allocation is very fast and simple: just increment the `alloc` pointer. No free list needed.

Collection is relatively cheap, especially if there is a lot of garbage. This is because **only reachable objects are touched**. Does not touch garbage. (Mark & Sweep touches every object in memory.)

Stop & Copy also improves **memory locality**, which can increase program execution speed.

Disadvantages: only uses half heap, and pointers get rewritten (prevents e.g. adress arithmetic).

Key trade-off of GC

GC **effortlessly** prevents many serious and hard to track down bugs, hence leads to much nicer and more readable program structure. So in most cases programs in a GCed language will be easier to write and maintain than corresponding programs in non GCed languages.

Key trade-off of GC

GC **effortlessly** prevents many serious and hard to track down bugs, hence leads to much nicer and more readable program structure. So in most cases programs in a GCed language will be easier to write and maintain than corresponding programs in non GCed languages.

But reduces programmer control, e.g. data layout in memory, or when memory is deallocated. Sometimes (but not very often) this is a problem.

Key trade-off of GC

GC **effortlessly** prevents many serious and hard to track down bugs, hence leads to much nicer and more readable program structure. So in most cases programs in a GCed language will be easier to write and maintain than corresponding programs in non GCed languages.

But reduces programmer control, e.g. data layout in memory, or when memory is deallocated. Sometimes (but not very often) this is a problem.

The GC (or OS kernel) itself has to be written in a language that doesn't have GC!

GC pauses

GC is extremely effective at avoiding the problems of manual memory managment, at the cost of some loss of performance.

GC pauses

GC is extremely effective at avoiding the problems of manual memory managment, at the cost of some loss of performance.

Some applications need to be **responsive**, e.g. to user input.

GC pauses

GC is extremely effective at avoiding the problems of manual memory managment, at the cost of some loss of performance.

Some applications need to be **responsive**, e.g. to user input. Simple implementations of GC reclaim the whole (mark & sweep) or half of the whole (stop & copy) heap in one go. While they operate, the computation of the user's program is suspended.

GC pauses

GC is extremely effective at avoiding the problems of manual memory managment, at the cost of some loss of performance.

Some applications need to be **responsive**, e.g. to user input. Simple implementations of GC reclaim the whole (mark & sweep) or half of the whole (stop & copy) heap in one go. While they operate, the computation of the user's program is suspended.

The bigger the heap, the longer the pause.

GC pauses

GC is extremely effective at avoiding the problems of manual memory managment, at the cost of some loss of performance.

Some applications need to be **responsive**, e.g. to user input. Simple implementations of GC reclaim the whole (mark & sweep) or half of the whole (stop & copy) heap in one go. While they operate, the computation of the user's program is suspended.

The bigger the heap, the longer the pause.

Some applications cannot tolerate long pauses, and must always remain **responsive**, e.g. handle user input in at most 300 ms. Or if you want to deliver Javascript generated graphics in the browser at 60 FPS, say, you have 16 mSec per image.

GC pauses

GC is extremely effective at avoiding the problems of manual memory managment, at the cost of some loss of performance.

Some applications need to be **responsive**, e.g. to user input. Simple implementations of GC reclaim the whole (mark & sweep) or half of the whole (stop & copy) heap in one go. While they operate, the computation of the user's program is suspended.

The bigger the heap, the longer the pause.

Some applications cannot tolerate long pauses, and must always remain **responsive**, e.g. handle user input in at most 300 ms. Or if you want to deliver Javascript generated graphics in the browser at 60 FPS, say, you have 16 mSec per image.

Variants of basic GC algorithms have been developed that allow GCed programs to remain responsive.

GC pauses

Variants of basic GC algorithms have been developed that allow GCed programs to remain responsive.

GC pauses

Variants of basic GC algorithms have been developed that allow GCed programs to remain responsive.

- **Concurrent GC** which happens in the background without stopping the program.

GC pauses

Variants of basic GC algorithms have been developed that allow GCed programs to remain responsive.

- **Concurrent GC** which happens in the background without stopping the program.
- **Incremental GC** Incremental GC reclaims heap memory is small increments. Rather than doing a full GC cycle when running out of memory, small parts of the heap are GCed regularly.

GC pauses

Variants of basic GC algorithms have been developed that allow GCed programs to remain responsive.

- **Concurrent GC** which happens in the background without stopping the program.
- **Incremental GC** Incremental GC reclaims heap memory is small increments. Rather than doing a full GC cycle when running out of memory, small parts of the heap are GCed regularly.
- **Generational GC** can be seen as a variant of incremental GC. It's widely used, e.g. in the JVM, Python (CPython, PyPy), Chrome's V8 Javascript JIT compiler, and probably a whole lot more.

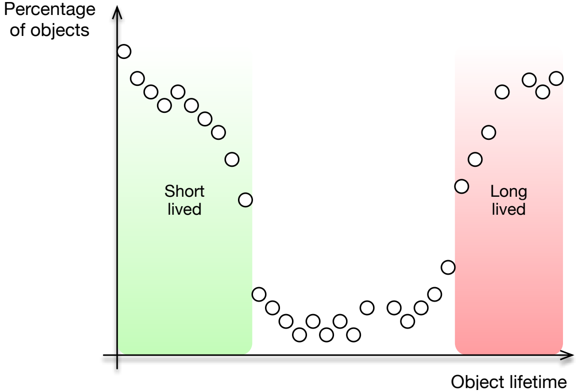
GC pauses

Variants of basic GC algorithms have been developed that allow GCed programs to remain responsive.

- **Concurrent GC** which happens in the background without stopping the program.
- **Incremental GC** Incremental GC reclaims heap memory is small increments. Rather than doing a full GC cycle when running out of memory, small parts of the heap are GCed regularly.
- **Generational GC** can be seen as a variant of incremental GC. It's widely used, e.g. in the JVM, Python (CPython, PyPy), Chrome's V8 Javascript JIT compiler, and probably a whole lot more. Let's look at generational GC in more detail.

Generational GC

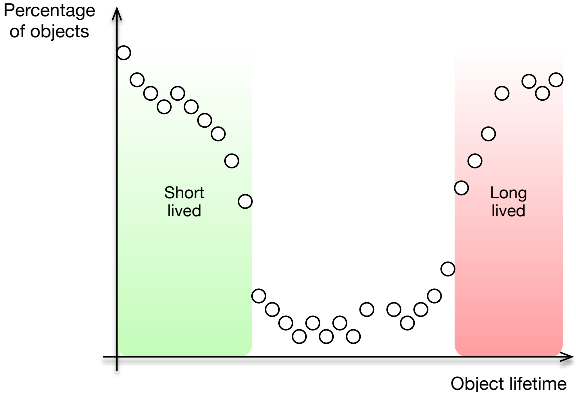
Generational GC is based on the following empirical insight about the lifetimes of objects in the heap.



Objects in the heap are usually either really short lived, or really long lived.

Generational GC

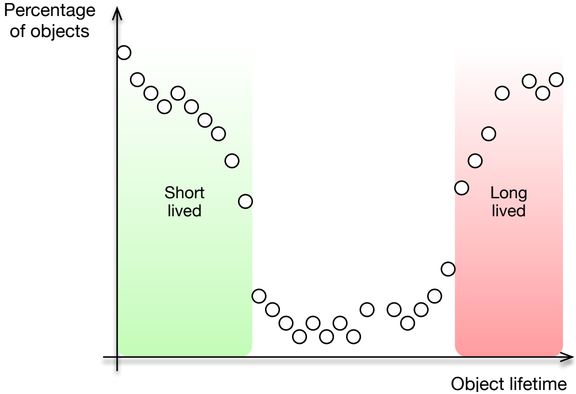
Generational GC is based on the following empirical insight about the lifetimes of objects in the heap.



Objects in the heap are usually either really short lived, or really long lived. In GC terms: if an object has survived a few GC cycles, then it is likely to be long-lived and survive more GC cyles.

Generational GC

Generational GC is based on the following empirical insight about the lifetimes of objects in the heap.



Objects in the heap are usually either really short lived, or really long lived. In GC terms: if an object has survived a few GC cycles, then it is likely to be long-lived and survive more GC cyles. How can we use this insight to improve GC?

Generational GC

Generational GC uses **several heaps**, e.g.:

Generational GC

Generational GC uses **several heaps**, e.g.:

- One heap for **short-lived** objects.

Generational GC

Generational GC uses **several heaps**, e.g.:

- One heap for **short-lived** objects.
- One heap for **long-lived** objects.

Generational GC

Generational GC uses **several heaps**, e.g.:

- One heap for **short-lived** objects.
- One heap for **long-lived** objects.

How do these heaps relate to each other?

Generational GC

Generational GC uses **several heaps**, e.g.:

- One heap for **short-lived** objects.
- One heap for **long-lived** objects.

How do these heaps relate to each other?

- Objects are first **allocated** in the heap for **short-lived objects**.

Generational GC

Generational GC uses **several heaps**, e.g.:

- One heap for **short-lived** objects.
- One heap for **long-lived** objects.

How do these heaps relate to each other?

- Objects are first **allocated** in the heap for **short-lived objects**.
- If an object **survives a couple of collections** in the heap for short-lived objects, it is **moved** to the heap for long-lived objects.
- GC is typically run only on the heap for short-lived objects. As there are many short-lived objects, GC is likely to reclaim memory just from that heap.
- If GC on the heap for short-lived objects fails, the heap for long-lived objects is GCed. This is rare!

Generational GC

Generational GC uses **several heaps**, e.g.:

- One heap for **short-lived** objects.
- One heap for **long-lived** objects.

How do these heaps relate to each other?

- Objects are first **allocated** in the heap for **short-lived objects**.
- If an object **survives a couple of collections** in the heap for short-lived objects, it is **moved** to the heap for long-lived objects.
- GC is typically run only on the heap for short-lived objects. As there are many short-lived objects, GC is likely to reclaim memory just from that heap.
- If GC on the heap for short-lived objects fails, the heap for long-lived objects is GCed. This is rare!

Since moving objects is a requirement for generational GC, it's especially suitable for combination with Stop & Copy GC.

Generational GC in the JVM

Generational GC in the JVM

The choice of GC is really important for performance for memory intensive computation. A large amount of effort has been spent on improving the JVM GCs.

Generational GC in the JVM

The choice of GC is really important for performance for memory intensive computation. A large amount of effort has been spent on improving the JVM GCs.

As of December 2019, the JVM ships with at 7 (!) different garbage collectors. Most are **generational** collectors, using several heaps (called generations) with complicated rules how to move between them:

- **Young** Generation. This is subdivided into three (!) sub-heaps: **Eden** (where objects are created), and two **survivor** spaces.
- **Old** Generation.
- **Permanent** Generation.

Generational GC in the JVM

The choice of GC is really important for performance for memory intensive computation. A large amount of effort has been spent on improving the JVM GCs.

As of December 2019, the JVM ships with at 7 (!) different garbage collectors. Most are **generational** collectors, using several heaps (called generations) with complicated rules how to move between them:

- **Young** Generation. This is subdivided into three (!) sub-heaps: **Eden** (where objects are created), and two **survivor** spaces.
- **Old** Generation.
- **Permanent** Generation.

Note that these complicated GC structures are the result of intensive empirical investigations into memory usage by real-world programs.

GC in the JVM

The permanent heap is used for objects that the JVM finds convenient to have the garbage collector manage, such as objects describing classes and methods, as well as the classes and methods (method tables) themselves.

GC in the JVM

The permanent heap is used for objects that the JVM finds convenient to have the garbage collector manage, such as objects describing classes and methods, as well as the classes and methods (method tables) themselves.

The young generation consists of Eden and two smaller survivor spaces. Most objects are initially allocated in Eden. (Some large objects may be allocated directly in the Old Generation heap.)

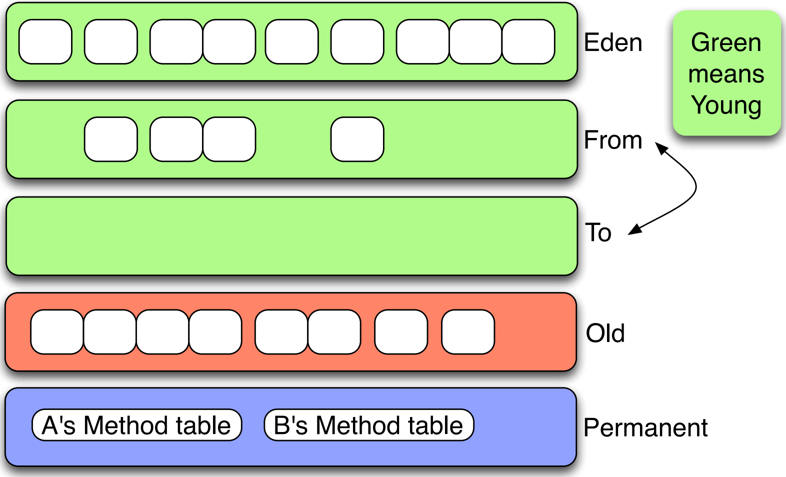
GC in the JVM

The permanent heap is used for objects that the JVM finds convenient to have the garbage collector manage, such as objects describing classes and methods, as well as the classes and methods (method tables) themselves.

The young generation consists of Eden and two smaller survivor spaces. Most objects are initially allocated in Eden. (Some large objects may be allocated directly in the Old Generation heap.)

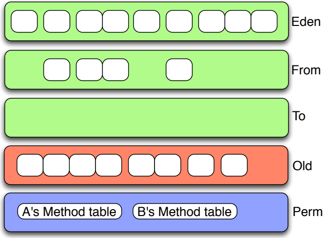
The survivor spaces hold objects that have survived at least one young generation collection and have thus been given additional chances to die before being considered "old enough" to be moved to the Old Generation. Survivor spaces holds such objects (we have 2 to do stop/copy with them).

Heap structure in the JVM (simplified)

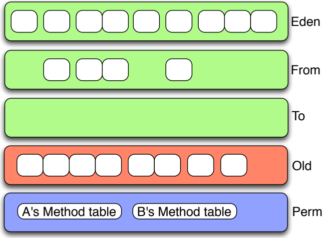


GC in the JVM

Objects are (mostly) created in Eden.



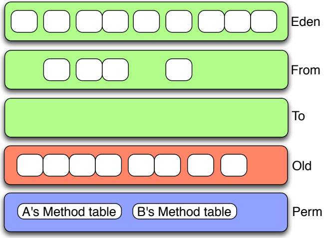
GC in the JVM



Objects are (mostly) created in Eden.

When the Young Generation fills up, a Young Generation collection (sometimes referred to as a minor collection) of just that generation is performed. When an Eden object survives on collection, it is moved to a Survivor heap (To or From). When an object has survived some number of collections in a Survivor heap, it is moved to the Old Generation heap.

GC in the JVM



Objects are (mostly) created in Eden.

When the Young Generation fills up, a Young Generation collection (sometimes referred to as a minor collection) of just that generation is performed. When an Eden object survives on collection, it is moved to a Survivor heap (To or From). When an object has survived some number of collections in a Survivor heap, it is moved to the Old Generation heap.

When the Old Generation fills up, a full collection (aka major collection) is done. Note that old generation doesn't have to be Stop & Copy since objects there are not moved.

Example `stanford-gc.pdf`

GC in the JVM

The JVM GC can be heavily tweaked by the programmer, e.g. size of the different generational heaps, collection strategies etc.

GC in the JVM

The JVM GC can be heavily tweaked by the programmer, e.g. size of the different generational heaps, collection strategies etc.

The JVM also offers extensive profiling features, so you can see exactly when and what kind of GC happens. This can be very useful if you are dealing with large amounts of data.

GC in the JVM

The JVM GC can be heavily tweaked by the programmer, e.g. size of the different generational heaps, collection strategies etc.

The JVM also offers extensive profiling features, so you can see exactly when and what kind of GC happens. This can be very useful if you are dealing with large amounts of data.

Example: `java -verbose:gc prog/Measure.java`

GC speed

"There is nothing difficult in GC, except to get it to run fast. That's 30-40 years of research." J. Vitek, personal communication, 2016.

GC speed

"There is nothing difficult in GC, except to get it to run fast. That's 30-40 years of research." J. Vitek, personal communication, 2016.

"Whenever your data increases by 10x, you have to redesign your algorithms from scratch" Programmer folklore.

GC speed

“There is nothing difficult in GC, except to get it to run fast. That’s 30-40 years of research.” J. Vitek, personal communication, 2016.

“Whenever your data increases by 10x, you have to redesign your algorithms from scratch” Programmer folklore.

As memory gets bigger, (as of 6. December 2019 the biggest memory you can rent on Amazon’s EC2 is **24 TB**) GC algorithms have to be, and are being rethought: GC, and memory management is an **active** area of research (see also Rust).

GC summary

GC automatises reclamation of unused heap memory (in C/C++ this is under programmer control). This tends to reduce the number of bugs.

GC summary

GC automatises reclamation of unused heap memory (in C/C++ this is under programmer control). This tends to reduce the number of bugs.

GC estimates object lifetime by reachability, starting from roots (registers and activation records on the stack), and then chasing pointers.

GC summary

GC automatises reclamation of unused heap memory (in C/C++ this is under programmer control). This tends to reduce the number of bugs.

GC estimates object lifetime by reachability, starting from roots (registers and activation records on the stack), and then chasing pointers.

Memory that is not reachable can be reclaimed.

GC summary

GC automatises reclamation of unused heap memory (in C/C++ this is under programmer control). This tends to reduce the number of bugs.

GC estimates object lifetime by reachability, starting from roots (registers and activation records on the stack), and then chasing pointers.

Memory that is not reachable can be reclaimed.

Many different techniques for reclamation of reachable memory, including copying.

GC summary

GC automatises reclamation of unused heap memory (in C/C++ this is under programmer control). This tends to reduce the number of bugs.

GC estimates object lifetime by reachability, starting from roots (registers and activation records on the stack), and then chasing pointers.

Memory that is not reachable can be reclaimed.

Many different techniques for reclamation of reachable memory, including copying.

Modern GC system tend to be generational (a collection of 'smaller' GCs) that can move live objects from one GC to another. The key assumption is that "most objects die young".

GC summary

GC automatises reclamation of unused heap memory (in C/C++ this is under programmer control). This tends to reduce the number of bugs.

GC estimates object lifetime by reachability, starting from roots (registers and activation records on the stack), and then chasing pointers.

Memory that is not reachable can be reclaimed.

Many different techniques for reclamation of reachable memory, including copying.

Modern GC system tend to be generational (a collection of ’smaller’ GCs) that can move live objects from one GC to another. The key assumption is that “most objects die young”.

Understanding GC is vital for high-performance programs, both for the normal programmer dealing with big data, and the compiler writer.

Last words

Last words

Compilers (and related subjects) great subject for final year projects.

Last words

Compilers (and related subjects) great subject for final year projects.

JRA (Junior Research Assistant) in the summer 2020.

Last words

Compilers (and related subjects) great subject for final year projects.

JRA (Junior Research Assistant) in the summer 2020.

Feel free to talk to me about this.


Merry
Christmas

Happy
Newyear