Runtime Environments

RECALL: *Intermediate Representations and Intermediate code generation*

- What is an intermediate representation?
- Why use an intermediate representation?
- Different types of Intermediate representations
  - AST, DAG, CFG, 3-address code
- 3-address code (assignment instructions, control-flow, functions,..)
- READ about (arrays, Switch,..)
- 3-address code Implementations (quadraples, triples,..)
- **Generating intermediate code**

Runtime Environments

- What is run-time support?
- **Parameter passing methods**
- **Storage allocation**
- **Activation records**
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

What is Run-time Support?

- Interfaces between the **program** and **computer system resources** are needed
  - There is a need to **manage memory** when a program is running
  - This **memory management** must connect to the data objects of programs
  - Programs request for memory blocks and release memory blocks
  - **Passing parameters** to functions needs attention
- Other resources such as printers, file systems, etc., also need to be accessed
- These are the main tasks of run-time support

Parameter passing methods

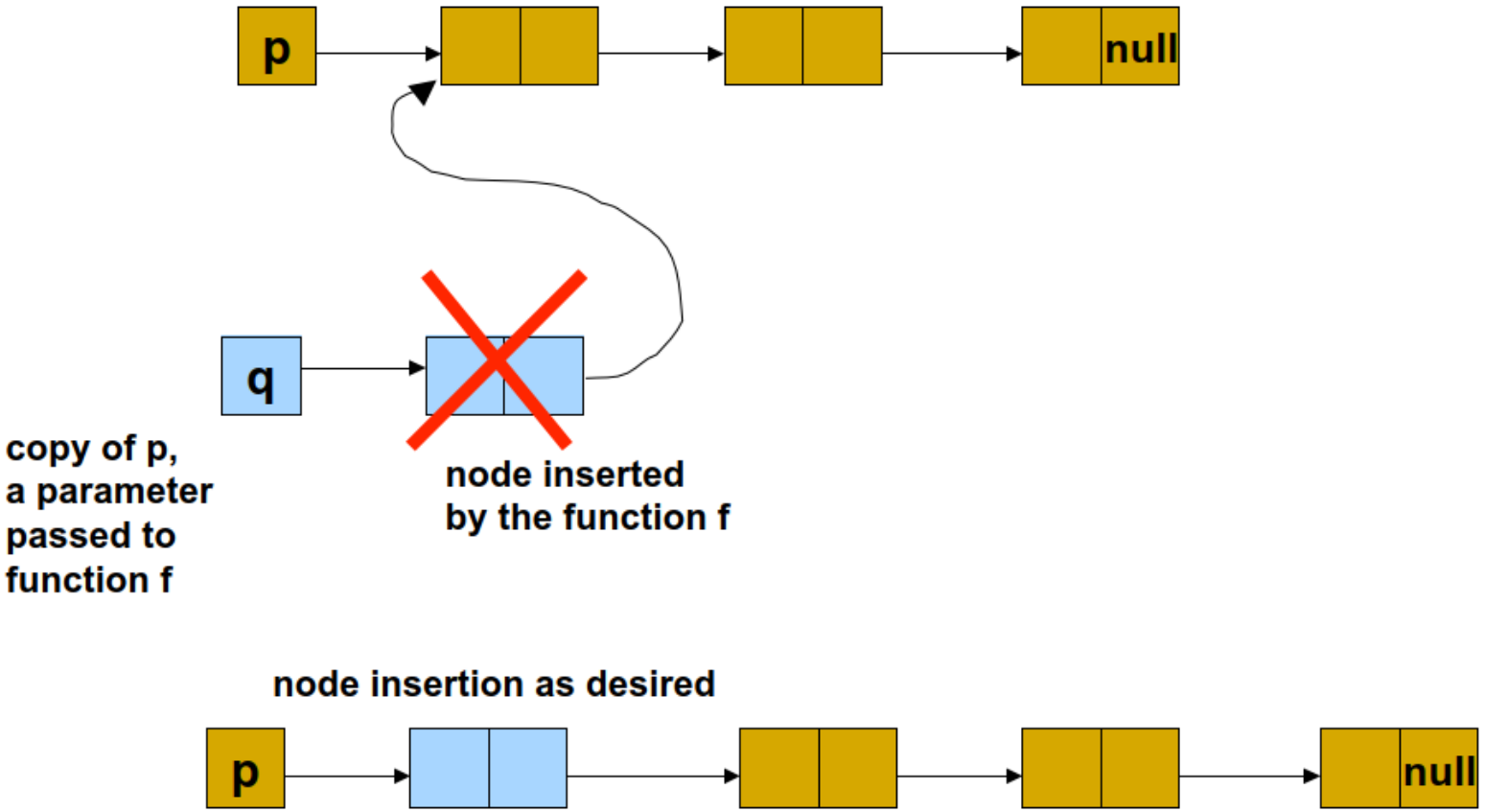
- Call-by-Value
- Call-by-Reference
- Call-by-Value-Result

Parameter passing methods (Call-by-Value)

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
  - Thus, there is **no way to change the actual parameters**.
  - Found in **C** and **C++**

  - C has only ***call-by-value*** method available
    - Passing pointers does not constitute *call-by-reference*
    - Pointers are also copied to another location
    - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using ***pointers to pointers***

Issue with Call-by-Value



Parameter passing methods (Call-by-Reference)

- At runtime, prior to the call, the parameter is evaluated and put in a *temporary location*, if it is not a variable
- The address of the variable (or the temporary) is passed to the called procedure

- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure

- Found in C++ and Java

Parameter passing methods (Call-by-Value-Result)

- Call-by-value-result is a hybrid of **Call-by-value** and **Call-by- reference**
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- **At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable**
- Becomes different from call-by-reference method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure
- Found in Ada

Example: Difference between Call-by-Value, Call-by-Reference, and Call-by-Value-Result

```
int a;
void Q()
    { a = a+1; }
void R(int x);
    { x = x+10; Q(); }
main()
    { a = 1; R(a); print(a); }
```

call-by-value	call-by-reference	call-by-value-result
2	**12**	**11**

Value of a printed

Note: In Call-by-V-R, value of x is copied into a, when proc R returns. Hence a=11.

Parameter passing methods (Call-by-Name)

- Use of a ***call-by-name*** parameter implies a textual substitution of the formal parameter name by the actual parameter

For example, if the procedure

```
void R (int X, int I);
{ I = 2; X = 5; I = 3; X = 1; }
```

is called by R(B[J*2], J)

this would result in (effectively) changing the body to

```
{ J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }
```

just before executing it

Parameter passing methods (Call-by-Name)

- Note that the actual parameter corresponding to **X** changes whenever **J** changes
  - Hence, we cannot evaluate the address of the actual parameter just once and use it
  - It **must be recomputed** every time we reference the formal parameter within the procedure

Example of Using the Four Parameter Passing Methods

```
1.  void swap (int x, int y)
2.  { int temp;
3.    temp = x;
4.    x = y;
5.    y = temp;
6.  } /*swap*/
7.  ...
8.  { i = 1;
9.    a[i] =10; /* int a[5]; */
10.  print(i,a[i]);
11.  swap(i,a[i]);
12.  print(i,a[1]); }
```

- Results from the 4 parameter passing methods (print statements)

call-by-value	call-by-reference	call-by-val-result	call-by-name
1 10 1 10	1 10 10 1	1 10 10 1	1 10 error!

Reason for the error in the Call-by-name Example

The problem is in the swap routine

temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */

- **What is run-time support?**
- **Parameter passing methods**
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Code and Data Area in Memory

- Most programming languages distinguish between **code** and **data**

- **Code** consists of only machine instructions and normally does not have embedded data
  - Code area normally does not grow or shrink in size as execution proceeds
    - Unless code is loaded dynamically or code is produced dynamically
    - Example– dynamic loading of classes
  - Memory area can be allocated to code statically
    - We will not consider dynamic loading of classes further

- **Data** area of a program may grow or shrink in size during execution

Static Versus Dynamic Storage Allocation

- Static allocation
  - Compiler makes the decision regarding storage allocation by looking ***only*** at the program text

- Dynamic allocation
  - Storage allocation decisions are made only ***while the program is running***
  - **Stack allocation**: Names local to a procedure are allocated space on a stack
  - **Heap allocation**
    - Used for data that may live even after a procedure call returns
    - Requires ***memory manager*** *with* ***garbage collection***

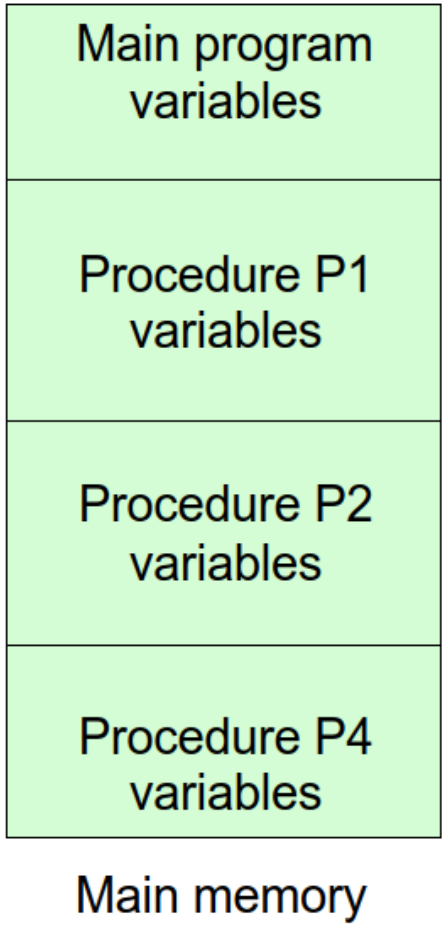
Static Data Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at **compile time**

  - No stack/heap allocation; no overheads
  - **Ex**: Fortran IV
  - Variable access is fast since addresses are known at compile time

Drawbacks?

- No recursion

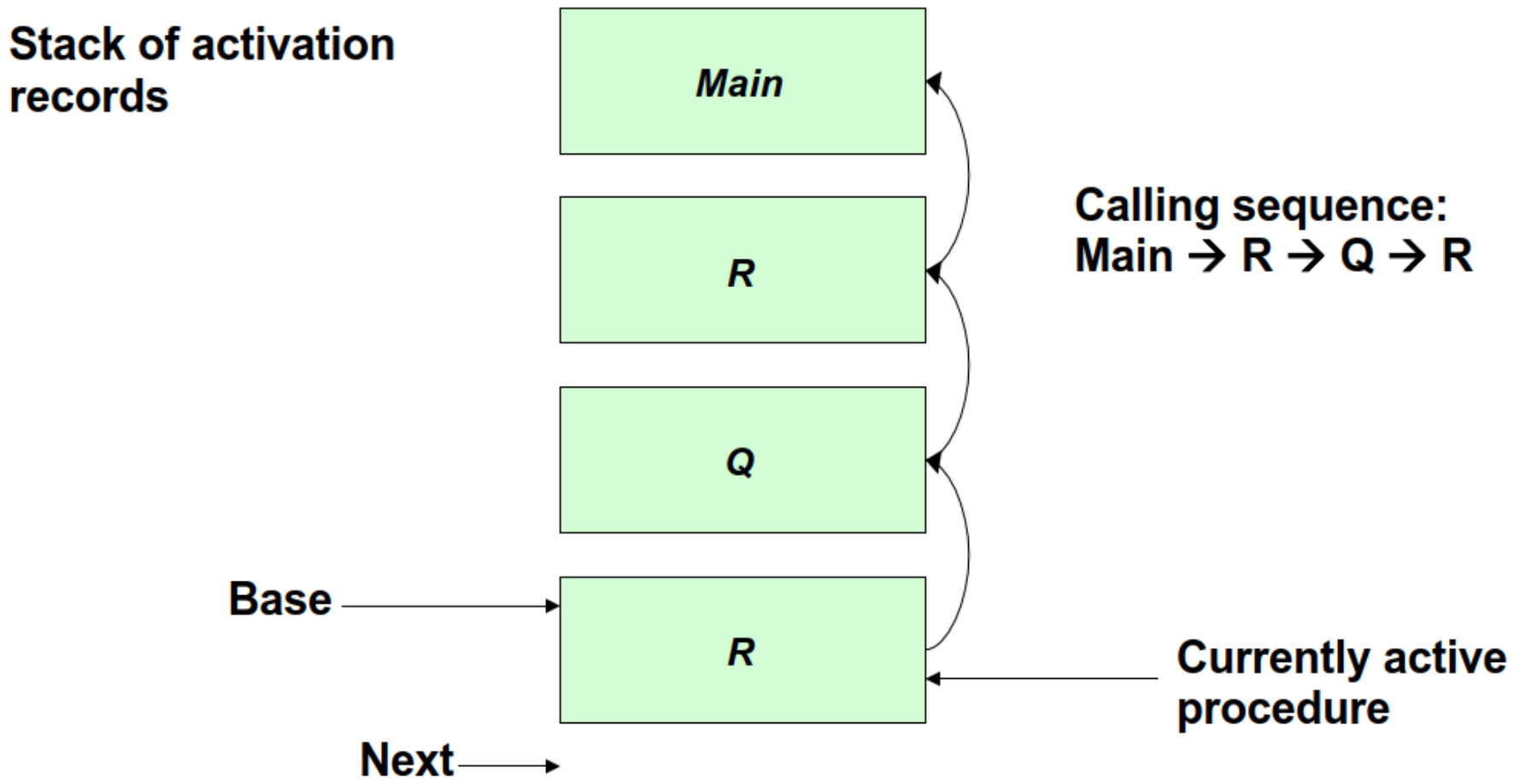


Dynamic Data Storage Allocation

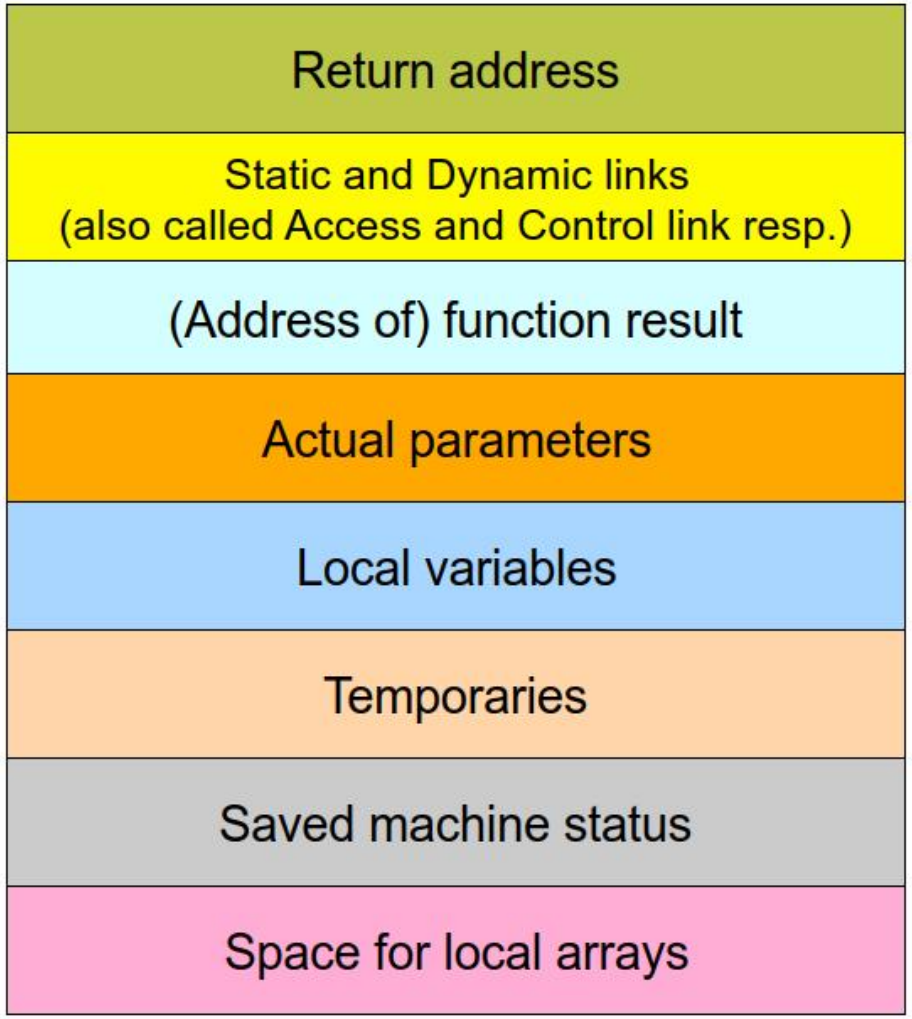
- Compiler allocates space only for **global variables** at **compile time**

- Space for **variables of procedures** will be allocated at **run-time**
  - Stack/heap allocation
  - **Ex**: C, C++, Java, Fortran 8/9
  - **Variable access is slow** (*compared to static allocation*) since addresses are accessed through the stack/heap pointer
  - **Recursion can be implemented**

Dynamic Stack Storage Allocation



Activation Record Structure



Note:

The position of the fields of the act. record as shown are only notional.

Implementations can choose different orders; e.g., function result could be after local var.

Variable Storage Offset Computation

- The compiler should compute
  - the offsets at which variables and constants will be stored in the activation record (**AR**)
- These offsets will be *with respect to* the pointer pointing to the beginning of the **AR**
- Variables are usually stored in the **AR** in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations

Overlapped Variable Storage for Blocks in C

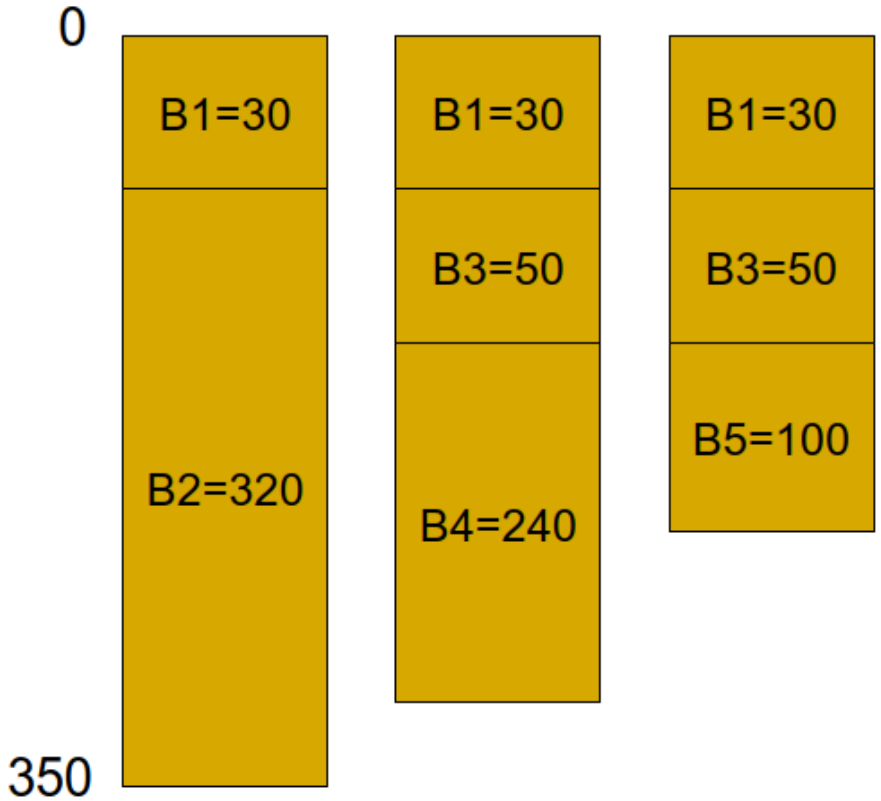
```
int example(int p1, int p2)
B1 { a,b,c; /* sizes - 10,10,10;
               offsets 0,10,20 */
     ...
     B2 { d,e,f; /* sizes - 100, 180, 40;
                    offsets 30, 130, 310 */
          ...}
     B3 { g,h,i; /* sizes - 20,20,10;
                    offsets 30, 50, 70 */
          ...
          B4 { j,k,l; /* sizes - 70, 150, 20;
                         offsets 80, 150, 300 */
               ... }
          B5 { m,n,p; /* sizes - 20, 50, 30;
                         offsets 80, 100, 150 */
               ... }
     }
}
```

Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
30+max(320, (50+240)) =
30+max(320,290) = 350

Overlapped storage

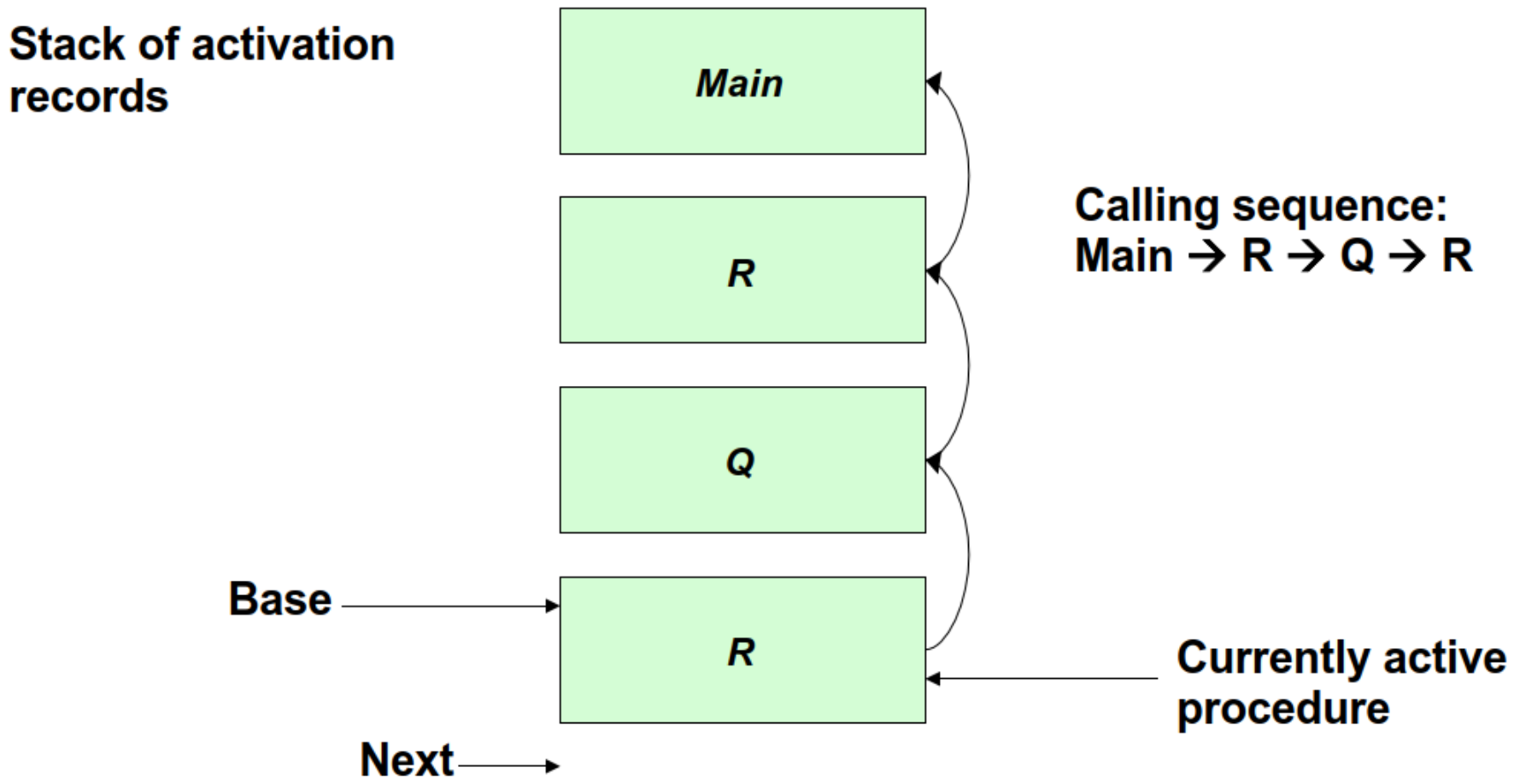
Overlapped storage

Overlapped Variable Storage for Blocks in C



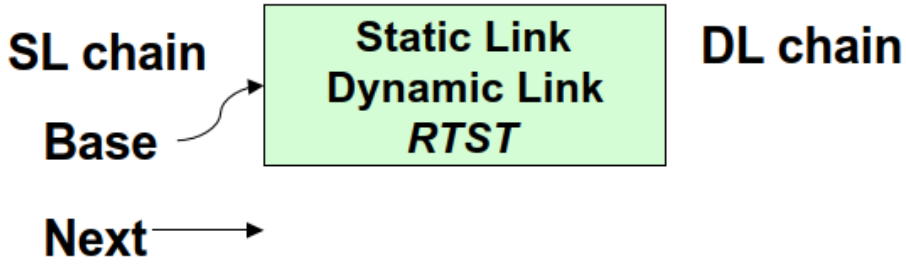
Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
30+max(320, (50+240)) =
30+max(320,290) = 350

Dynamic Stack Storage Allocation



Allocation of Activation Records (nested procedures)

```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```

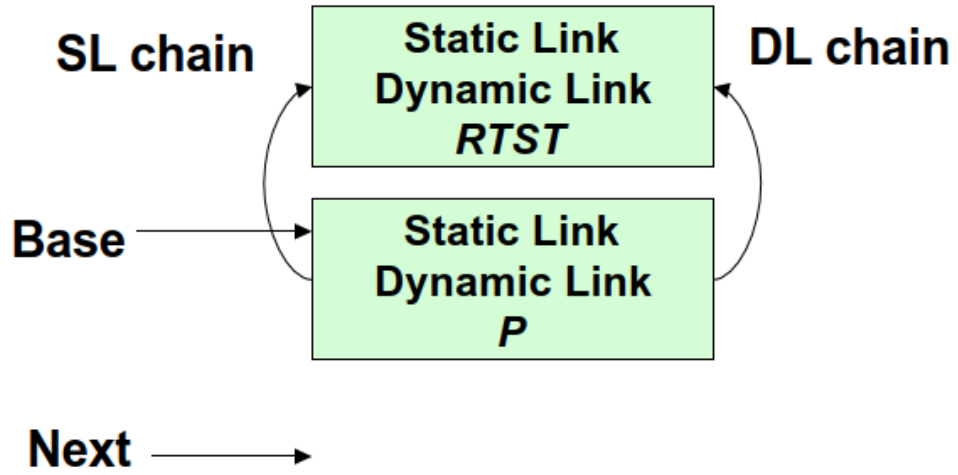


Activation records are created at procedure entry time and destroyed at procedure exit time

RTST -> P -> R -> Q -> R

Allocation of Activation Records (contd.)

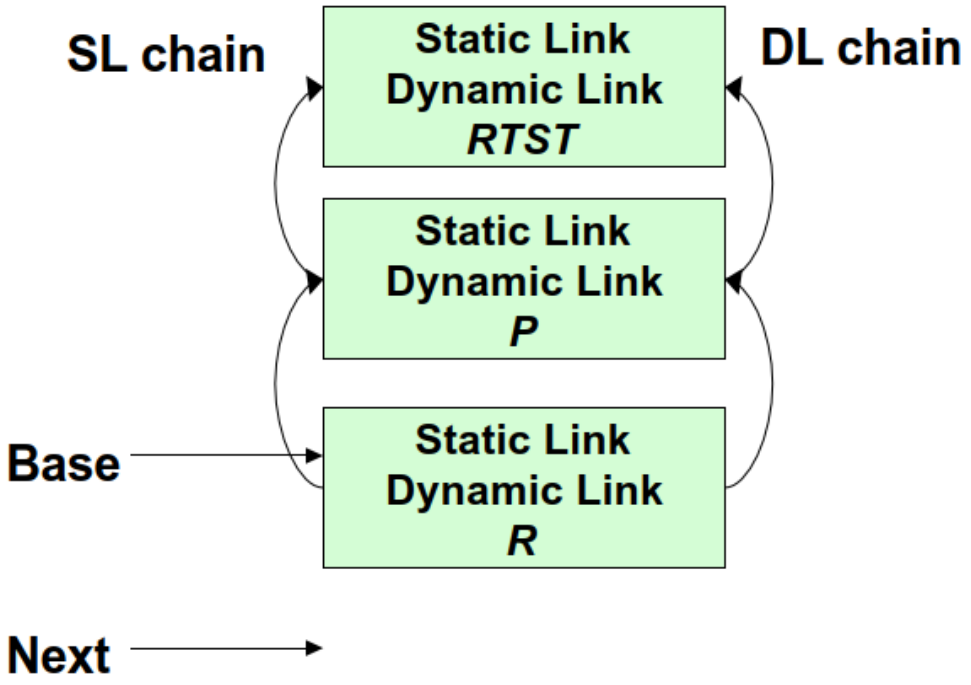
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> R -> Q -> R

Allocation of Activation Records (contd.)

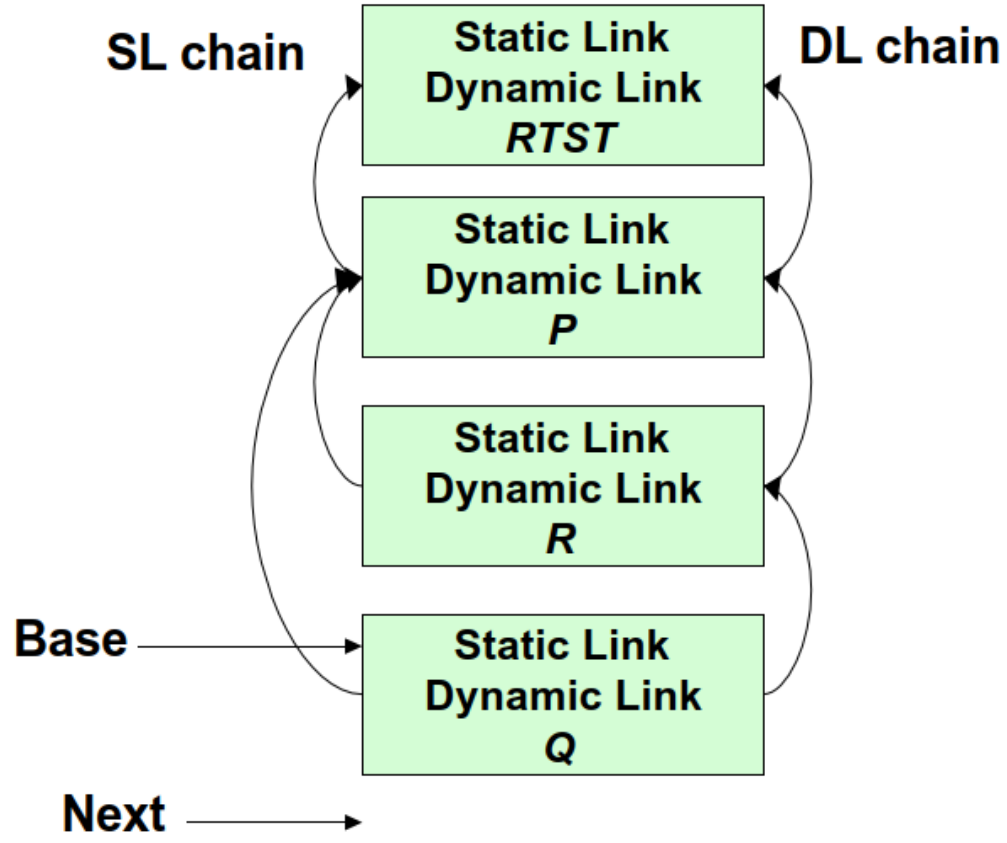
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> **R** -> Q -> R

Allocation of Activation Records (contd.)

```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```

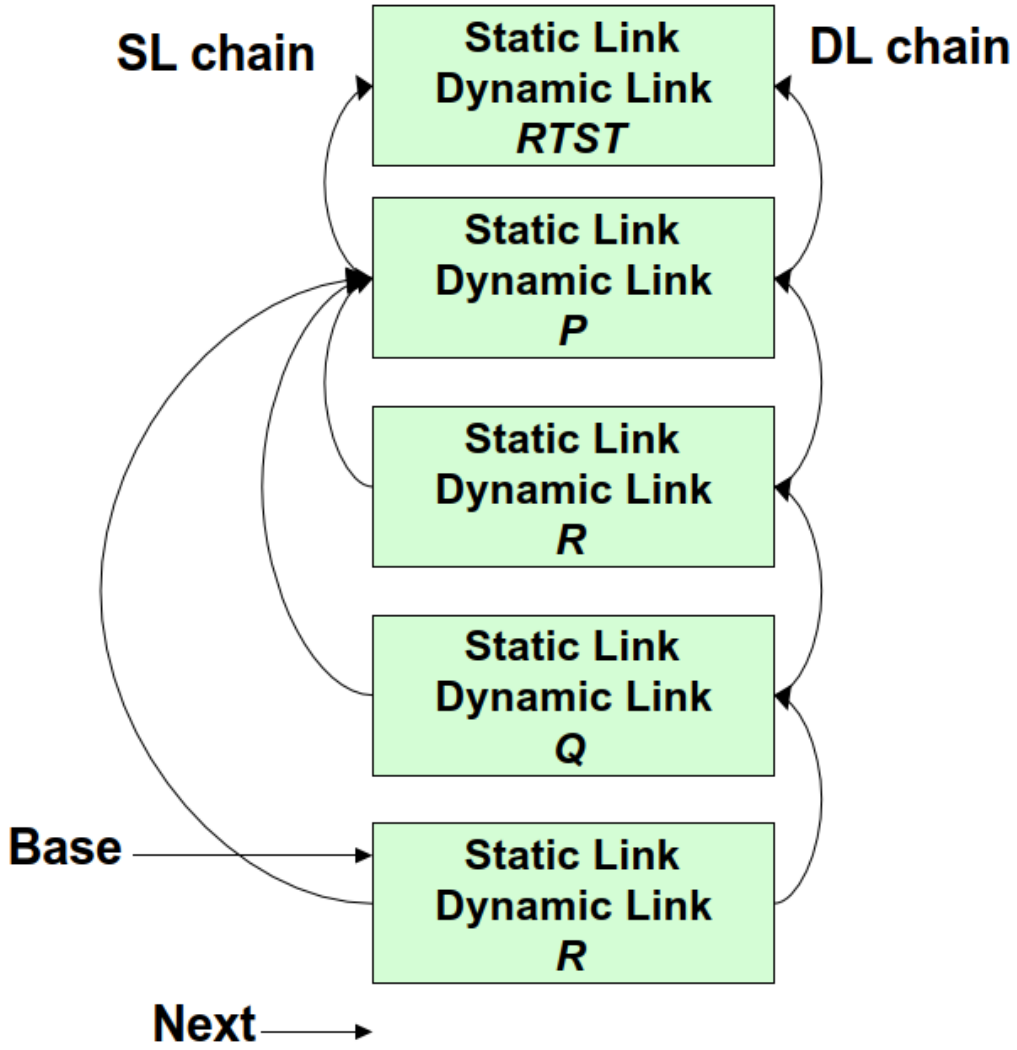


RTST -> **P** -> **R** -> **Q** -> R

Allocation of Activation Records (contd.)

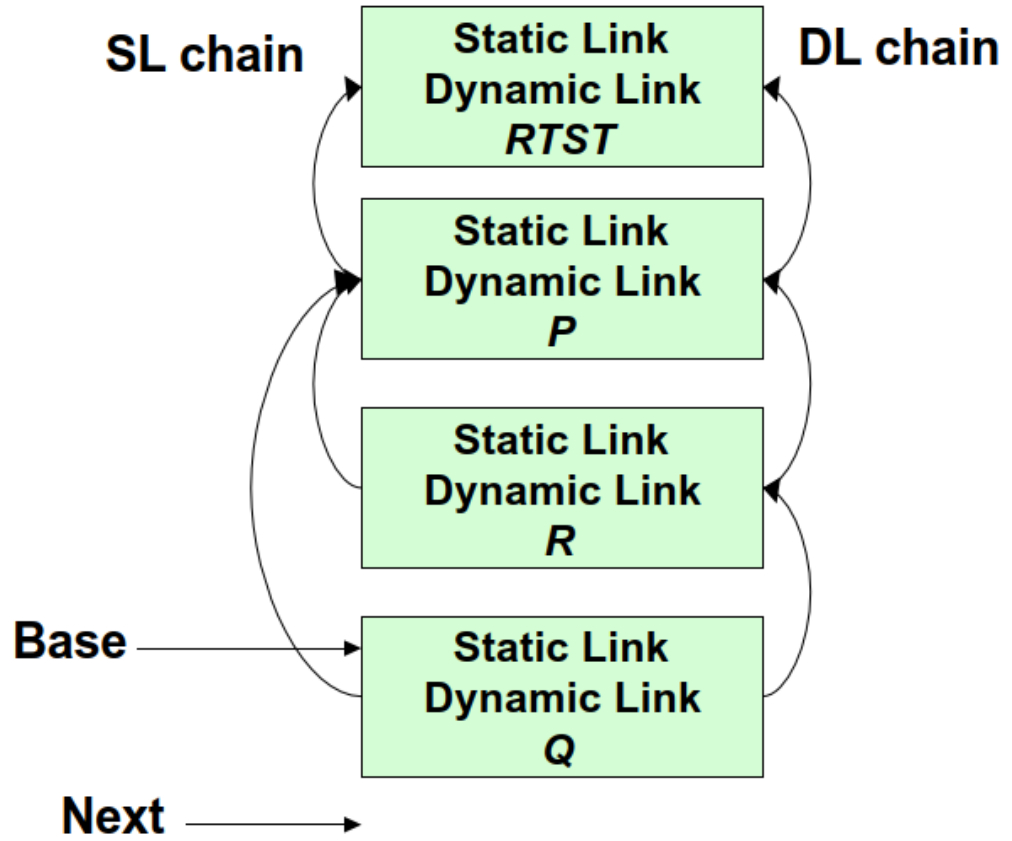
```
1  program RTST;
2    procedure P;
3      procedure Q;
         begin R; end
3      procedure R;
         begin Q; end
     begin R; end
   begin P; end
```

RTST[1] -> P[2] -> R[3] -> Q[3] -> R[3]



Allocation of Activation Records (contd.)

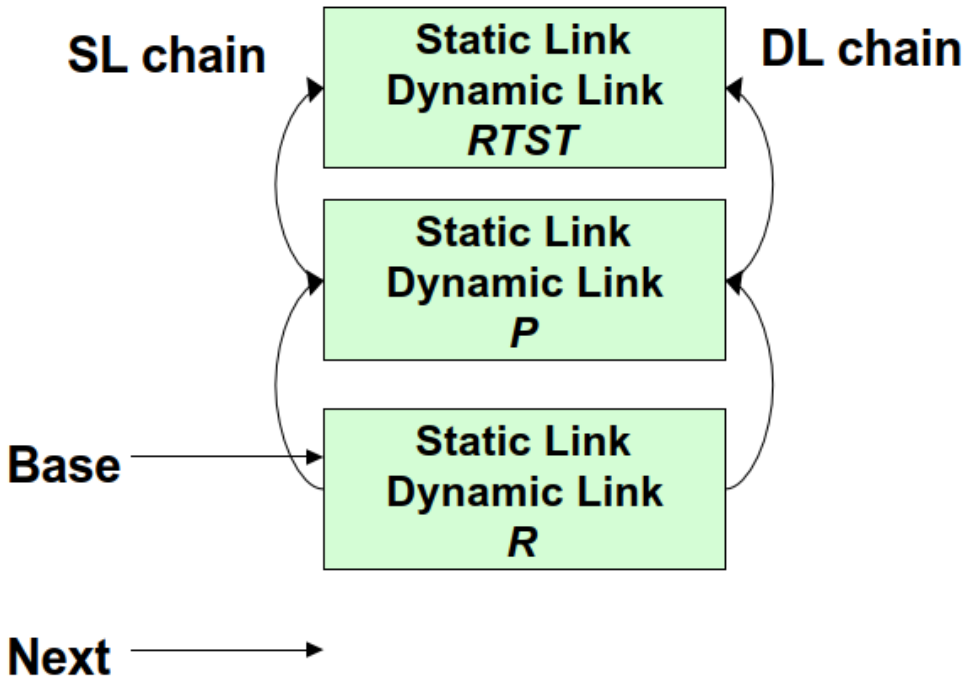
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> **R** -> **Q** <- R **Return from R**

Allocation of Activation Records (contd.)

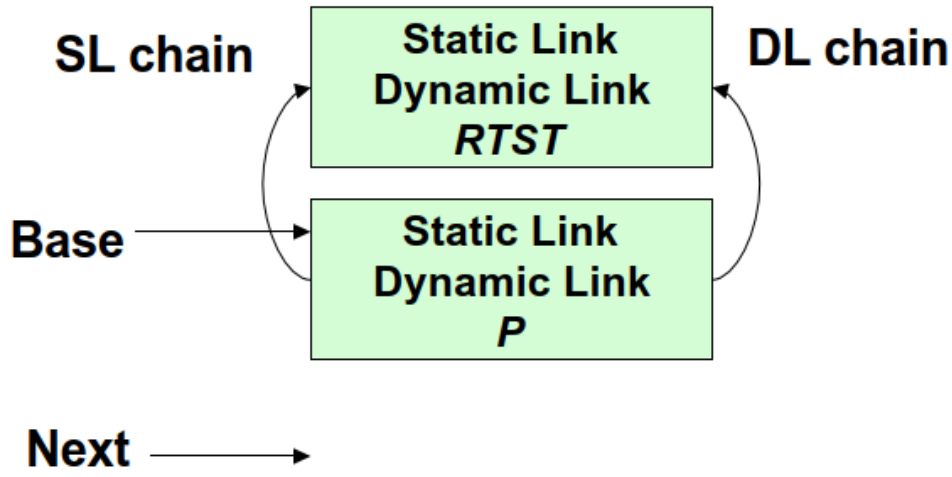
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> P -> R <- Q **Return from Q**

Allocation of Activation Records (contd.)

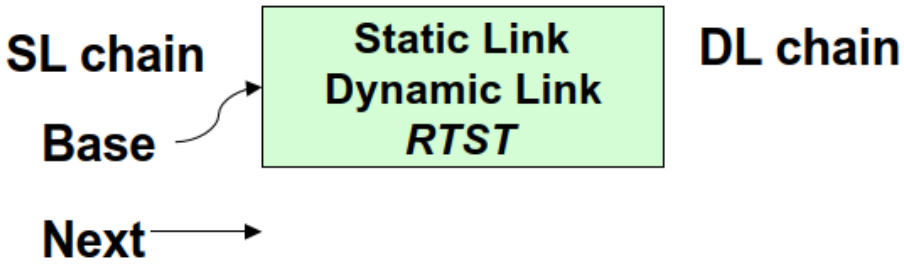
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> P <- R **Return from R**

Allocation of Activation Records (contd.)

```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



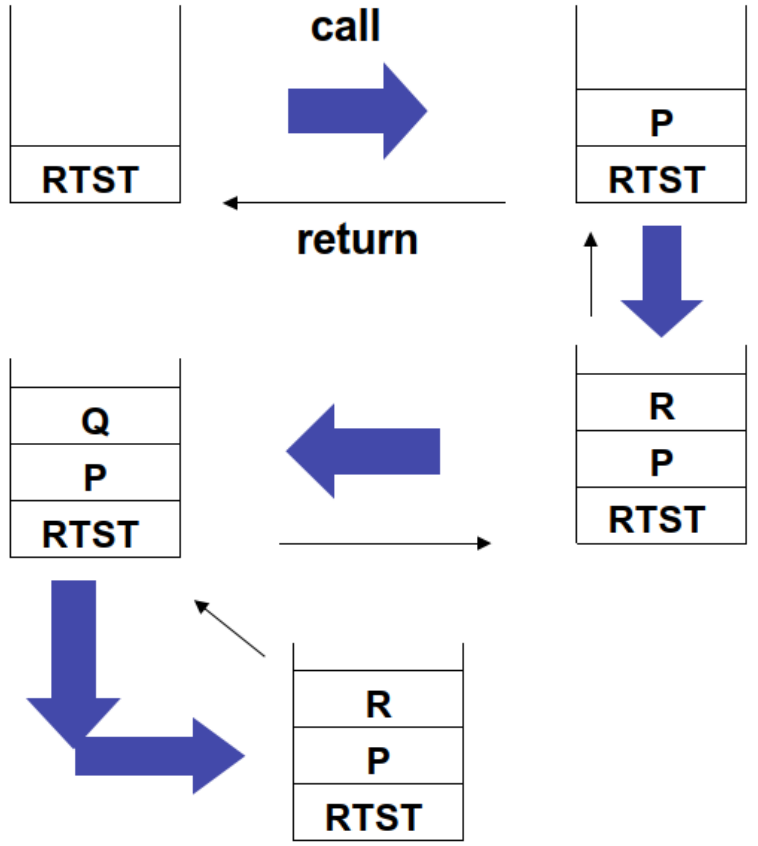
RTST <- P **Return from P**

Display Stack of Activation Records

```
1  program RTST;
2    procedure P;
3      procedure Q;
         begin R; end
3      procedure R;
         begin Q; end
     begin R; end
   begin P; end
```

Pop L_1-L_2+1 records off the display of the caller and push the pointer to AR of callee (L_1 – caller, L_2 – Callee)

The popped pointers are stored in the AR of the caller and restored to the DISPLAY after the callee returns



- **What is run-time support?**
- **Parameter passing methods**
- **Storage allocation**
- **Activation records**
- Static scope and dynamic scope
- Heap memory management
- Garbage Collection

Static Scope and Dynamic Scope

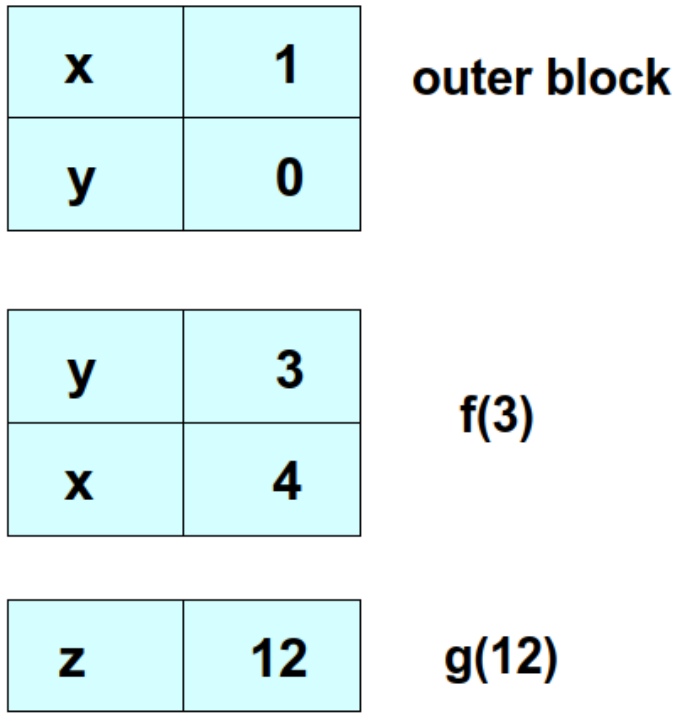
- *Static Scope*
  - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
  - Uses the *static* (unchanging) relationship between blocks in the program text
- *Dynamic Scope*
  - A global identifier refers to the identifier associated with the most recent activation record
  - Uses the actual sequence of calls that are executed in the *dynamic* (changing) execution of the program

- Both are identical as far as local variables are concerned

Static Scope and Dynamic Scope : An Example

```
int x = 1, y = 0;
int g(int z)
  { return x+z;}
int f(int y) {
    int x; x = y+1;
    return g(y*x);
}
y = f(3);
```

After the call to g,
Static scope: x = 1
Dynamic scope: x = 4



Stack of activation records after the call to *g*

Static Scope and Dynamic Scope: Another Example

```
float r = 0.25;
void show() { printf("%f",r); }
void small() {
   float r = 0.125; show();
}
int main (){
show(); small(); printf("\n");
show(); small(); printf("\n");
}
```

- Under static scoping, the output is

  0.25 0.25

  0.25 0.25

- Under dynamic scoping, the output is

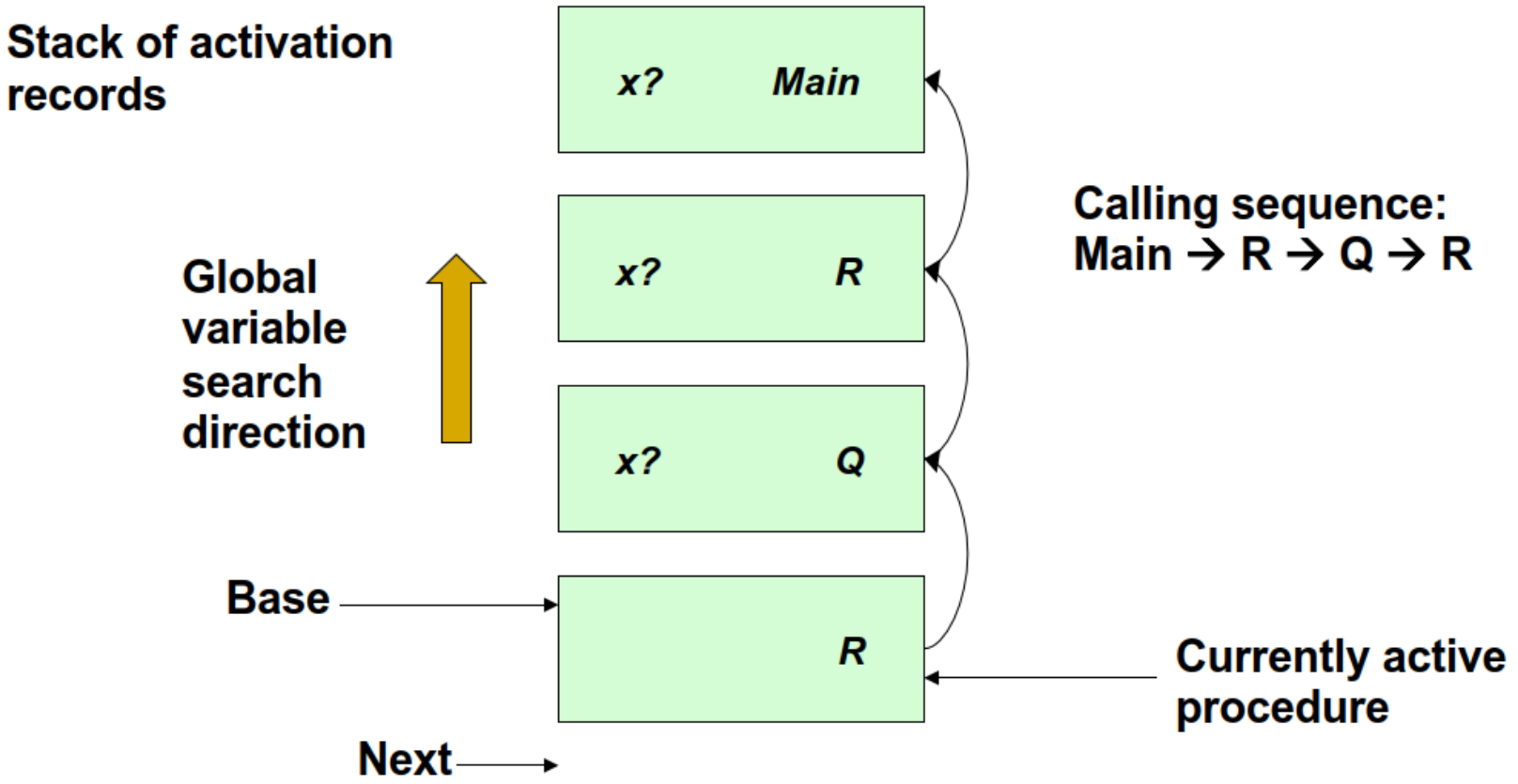
  0.25 0.125

  0.25 0.125

Implementing Dynamic Scope – Deep Access Method

- Use dynamic link
- Search activation records on the stack to find the first AR containing the ***non-local*** **name**
- **The depth of search depends on the input** to the program and **cannot be determined at compile time**

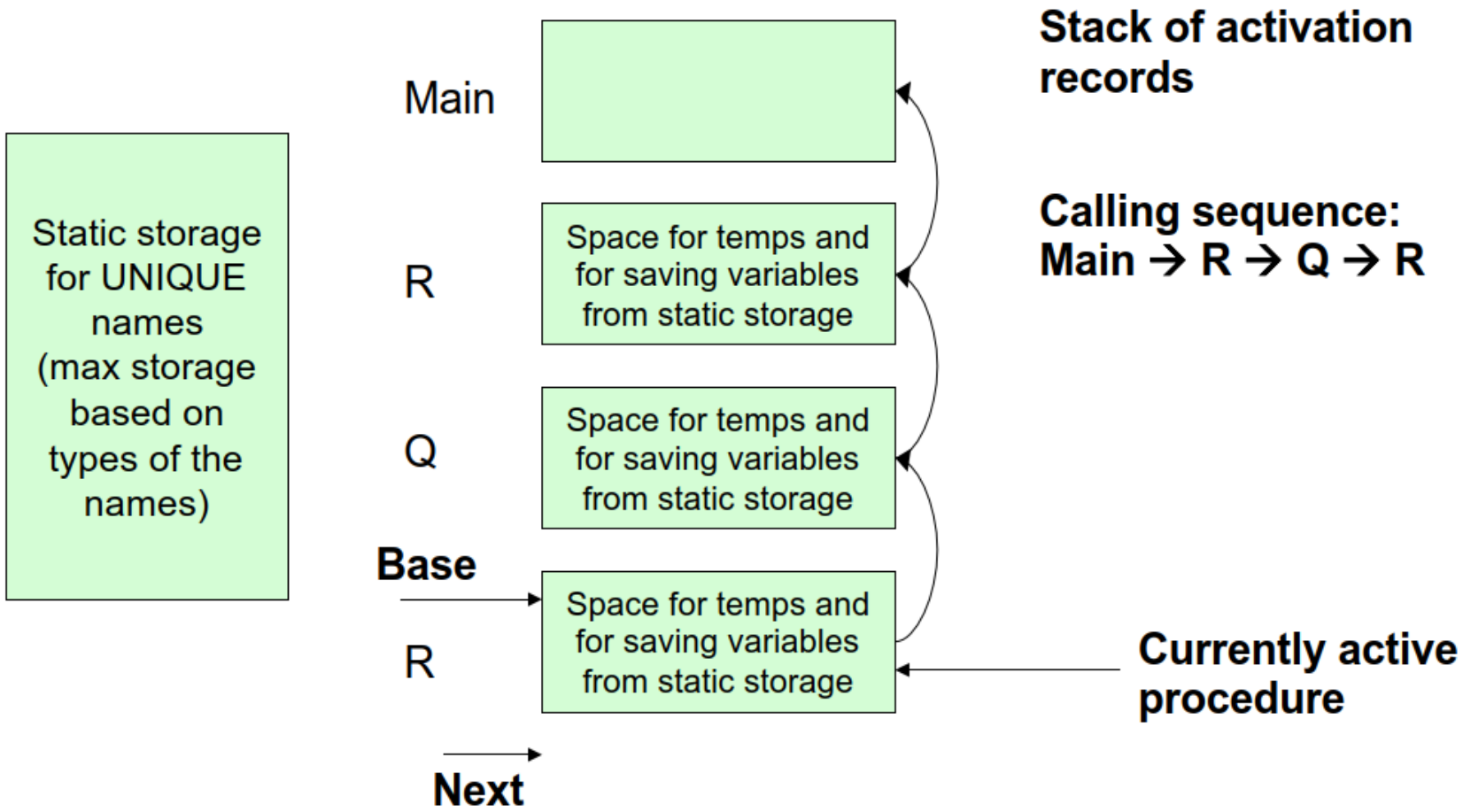
Deep Access Method - Example



Implementing Dynamic Scope – Shallow Access Method

- Allocate maximum static storage needed for ***each name*** (based on the types)
- When a new AR is created for a procedure p, a local name n in p takes over the static storage allocated to name n
  - Global (non-local) variables are also accessed from the static storage
  - Temporaries are located in the AR
- The previous value of n held in static storage is saved in the AR of **p** and is restored when the activation of p ends
- **Direct** and **quick** access to **globals**, but some overhead is incurred when activations begin and end

Shallow Access Method - Example



- **What is run-time support?**
- **Parameter passing methods**
- **Storage allocation**
- **Activation records**
- **Static scope and dynamic scope**
- Heap memory management
- Garbage Collection

Heap Memory Management

- Heap is used for allocating space for objects created at **run time**
  - For example: nodes of dynamic data structures such as **linked lists** and trees

- Dynamic memory allocation and deallocation based on the requirements of the program
  - ***malloc*()** and ***free*()** in C programs
  - ***new***() and ***delete***() in C++ programs
  - ***new***() and garbage collection in Java programs

- Allocation and deallocation may be completely manual (C/C++), semi-automatic (Java), or fully automatic (Lisp)

Memory Manager

- **Manages heap memory** by implementing mechanisms for allocation and deallocation, both manual and automatic

- **Goals**
  - Space efficiency: minimize fragmentation
  - Program efficiency: take advantage of locality of objects in memory and make the program run faster
  - Low overhead: allocation and deallocation must be efficient

- Heap is maintained either as a **doubly linked list** or as **bins** of free memory chunks

Allocation and Deallocation

- In the beginning, the heap is one large and contiguous block of memory
- As allocation requests are received, chunks are cut off from this block and given to the program
- As deallocations are made, chunks are returned to the heap and are free to be allocated again (holes)
- After a number of allocations and deallocations, memory becomes ***fragmented*** and is not contiguous
- Allocation from a fragmented heap may be made either in a ***first-fit*** or ***best-fit*** manner
- After a deallocation, we try to ***coalesce*** contiguous holes and make a bigger free chunk

First-Fit and Best-Fit Allocation Strategies

- The first-fit strategy picks the first available chunk that satisfies the allocation request
- The best-fit strategy searches and picks the smallest (best) possible chunk that satisfies the allocation request
- Both of them chop off a block of the required size from the chosen chunk, and return it to the program
- The rest of the chosen chunk remains in the heap

First-Fit and Best-Fit Allocation Strategies

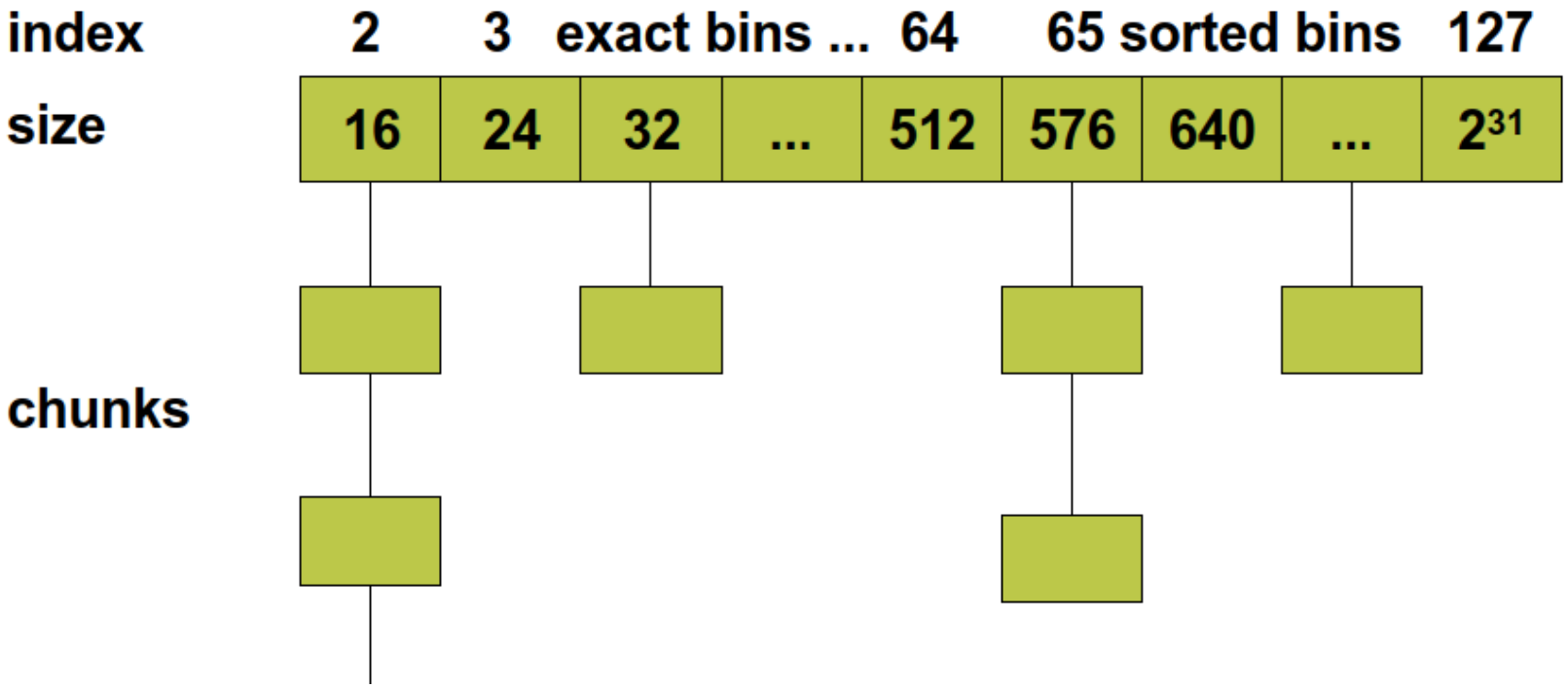
- **Best-fit** strategy has been shown to reduce fragmentation in practice, better than **first-fit** strategy

- Next-fit strategy tries to allocate the object in the chunk that has been split recently
  - Tends to ***improve speed of allocation***
  - Tends to ***improve spatial locality*** since objects allocated at about the same time tend to have ***similar reference patterns*** and life times (**cache behavior may be better**)

Bin-based Heap

- Free space is organized into **bins according to their sizes** (Memory Manager in GCC)
- Many more bins for smaller sizes, because there are many more small objects
- A bin for every multiple of 8-byte chunks (from 16 bytes to 512 bytes)
- Within each "**small size bin**", chunks are all of the **same size**
- In others, they are ordered by size
- The last chunk in the last bin is the ***wilderness chunk***, which gets us a chunk by going to the operating system

Bin-based Heap – An Example



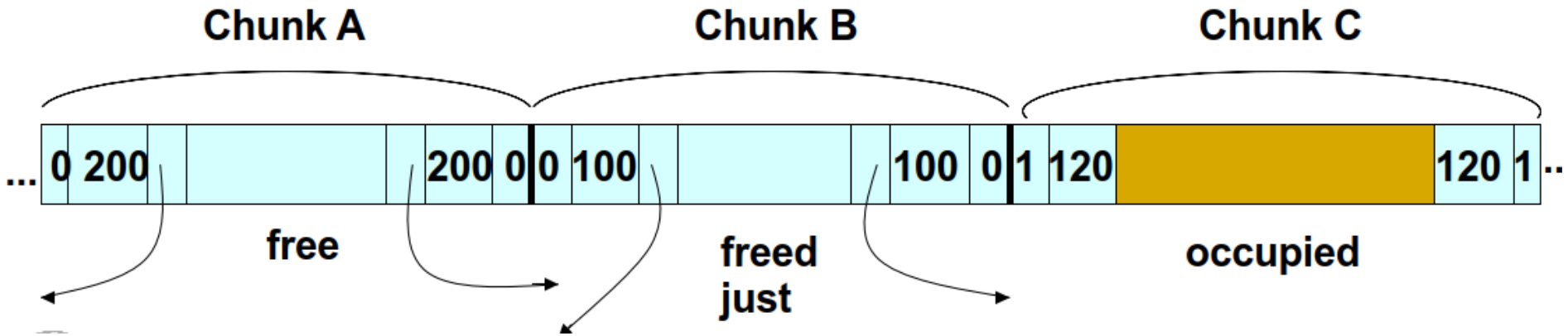
Managing and Coalescing Free Space

- Should coalesce adjacent chunks and reduce fragmentation
- Many small chunks together cannot hold one large object
- In the Lea memory manager, no coalescing in the **exact size** bins, only in the **sorted bins**

- **Boundary tags** (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
- A doubly linked list of free chunks

Boundary Tags and Doubly Linked List

3 adjacent chunks. Chunk B has been freed just now and returned to the free list. Chunks A and B can be merged, and this is done just before inserting it into the linked list. The merged chunk AB may have to be placed in a different bin.



- **What is run-time support?**
- **Parameter passing methods**
- **Storage allocation**
- **Activation records**
- **Static scope and dynamic scope**
- **Heap memory management**
- Garbage Collection

Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data

- Both are serious and hard to debug

- Solution: **automatic garbage collection**
  - **e.g, Reference Counting Garbage Collector**

Runtime Environments (Summary)

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Heap memory management
- Garbage Collection