Mid-term Summary

Programming Language Theory

Topics

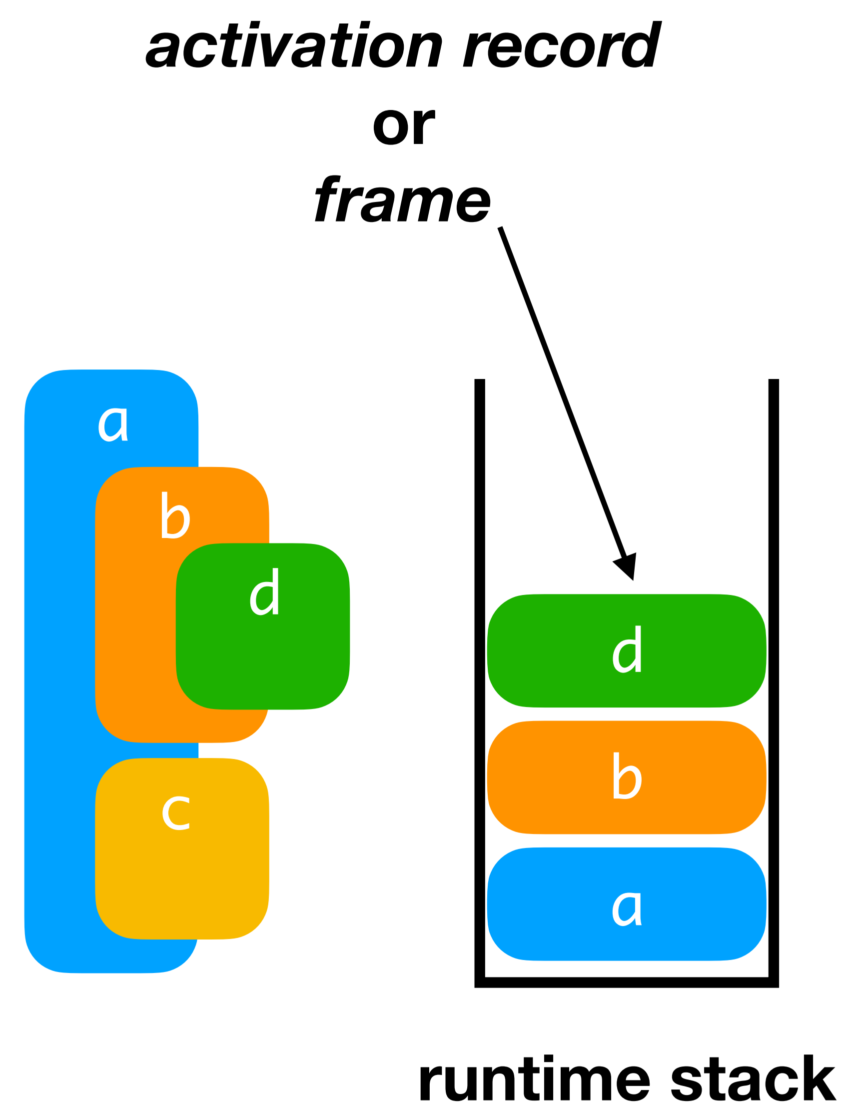
- BNF
- Parsing
- Static Memory Management
- **Dynamic Memory Managements**
- **Control Flow and Recursion**

Dynamic Memory Management

Dynamic Management w/ Stack

- We have already seen the basic concepts using stack.
- Each memory space allocated to a procedure activation (or an inline block) is called ***activation record*** or ***frame***.
- The stack containing activation records is called ***runtime stack***.

```
a(){
    b(){
        d(){
        }
        d();
    }
    c(){
    }
    b();
    c();
}
```



Activation Record and Dynamic Link

- An activation record with a local variable and intermediate results.

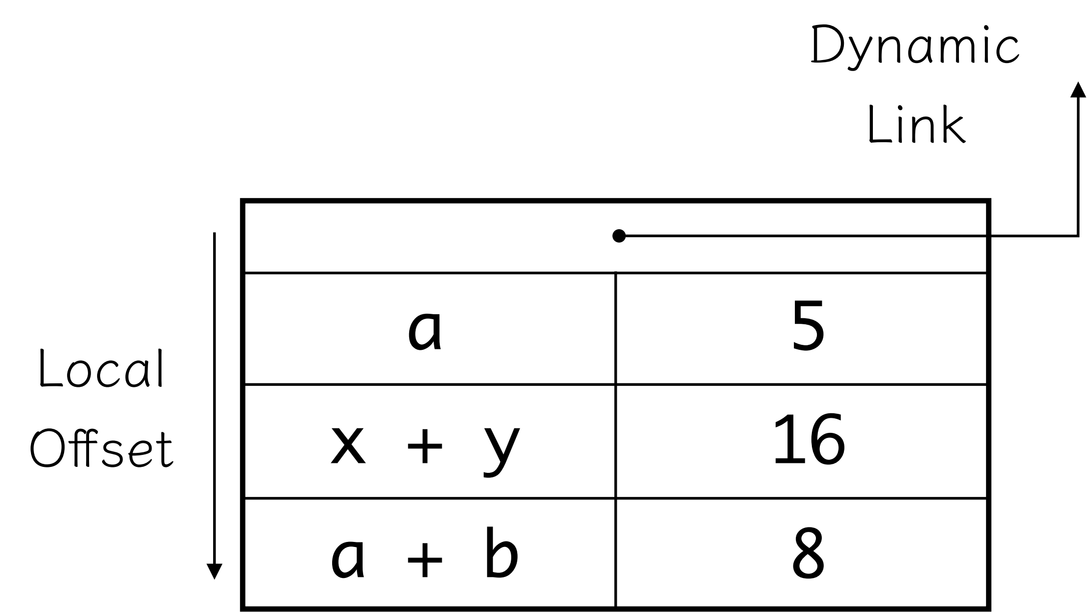
- **Dynamic Link** points to the start of *previous activation record* on the stack.

- This link is necessary since activation records have different sizes in general.

- From the start of activation record, we can use local offset to find a specific local variable.

In-line block

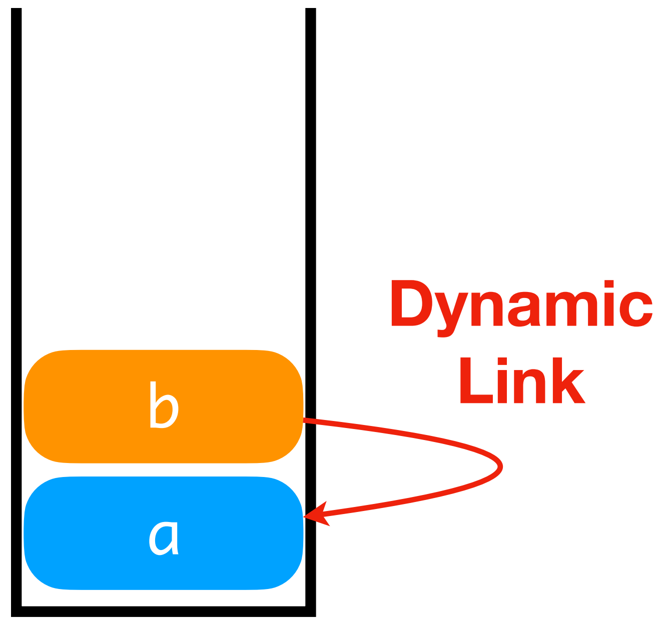
```
{
   int a = 5;
   b = (x+y) / (a+b);
}
```



Activation Record

Stack Management

- Activation records are stored to and removed from the stack at runtime.
- When ***procedure B is called by procedure A***, both ***A (caller)*** and ***B (callee)*** manage such operations on the stack.
  - For instance, an activation record of B should be pushed into the stack.
  - B's dynamic link points to A's activation record.

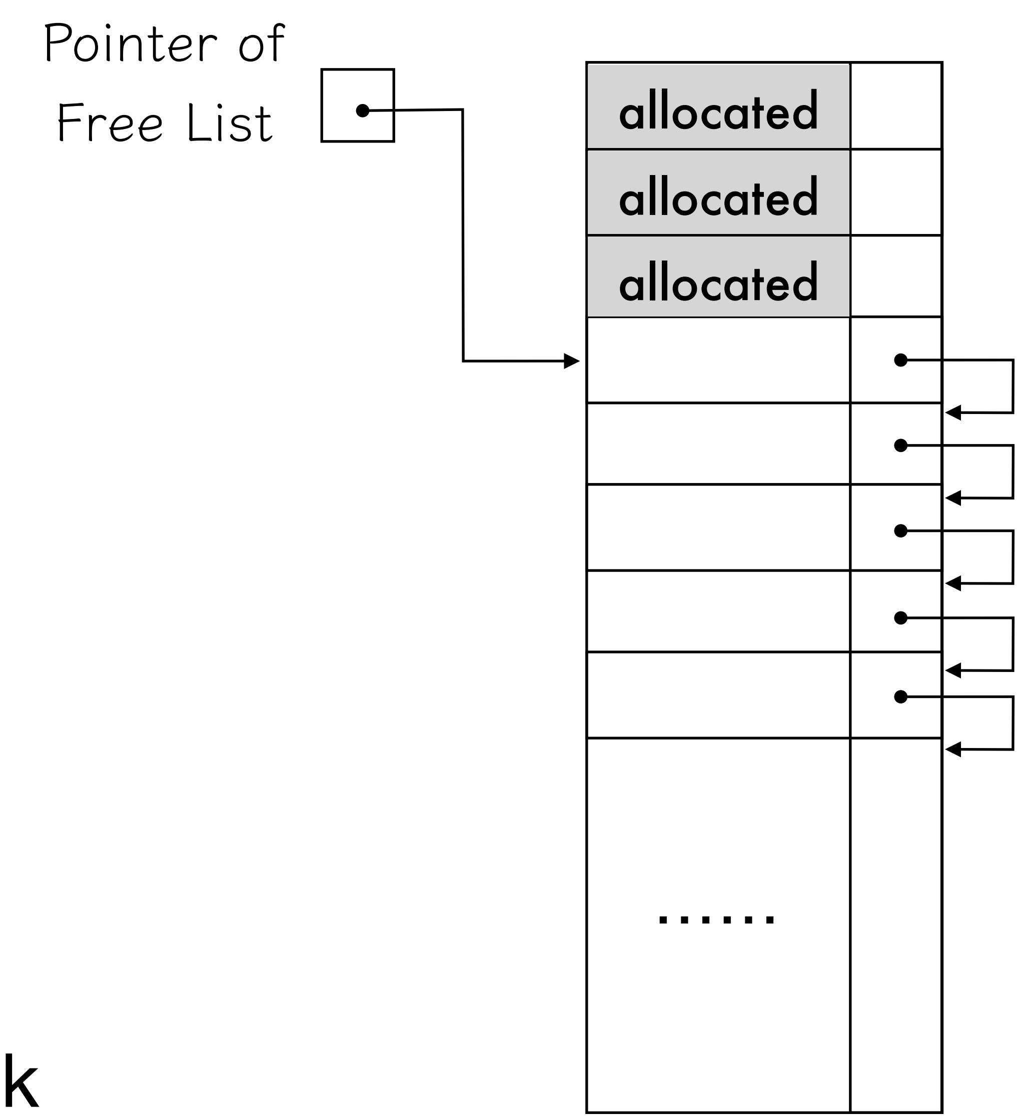


Heap Management

- Heap management methods fall into two main categories,
- based on whether the memory blocks are considered,
    - *Fixed Length Blocks*
    - *Variable Length Blocks*

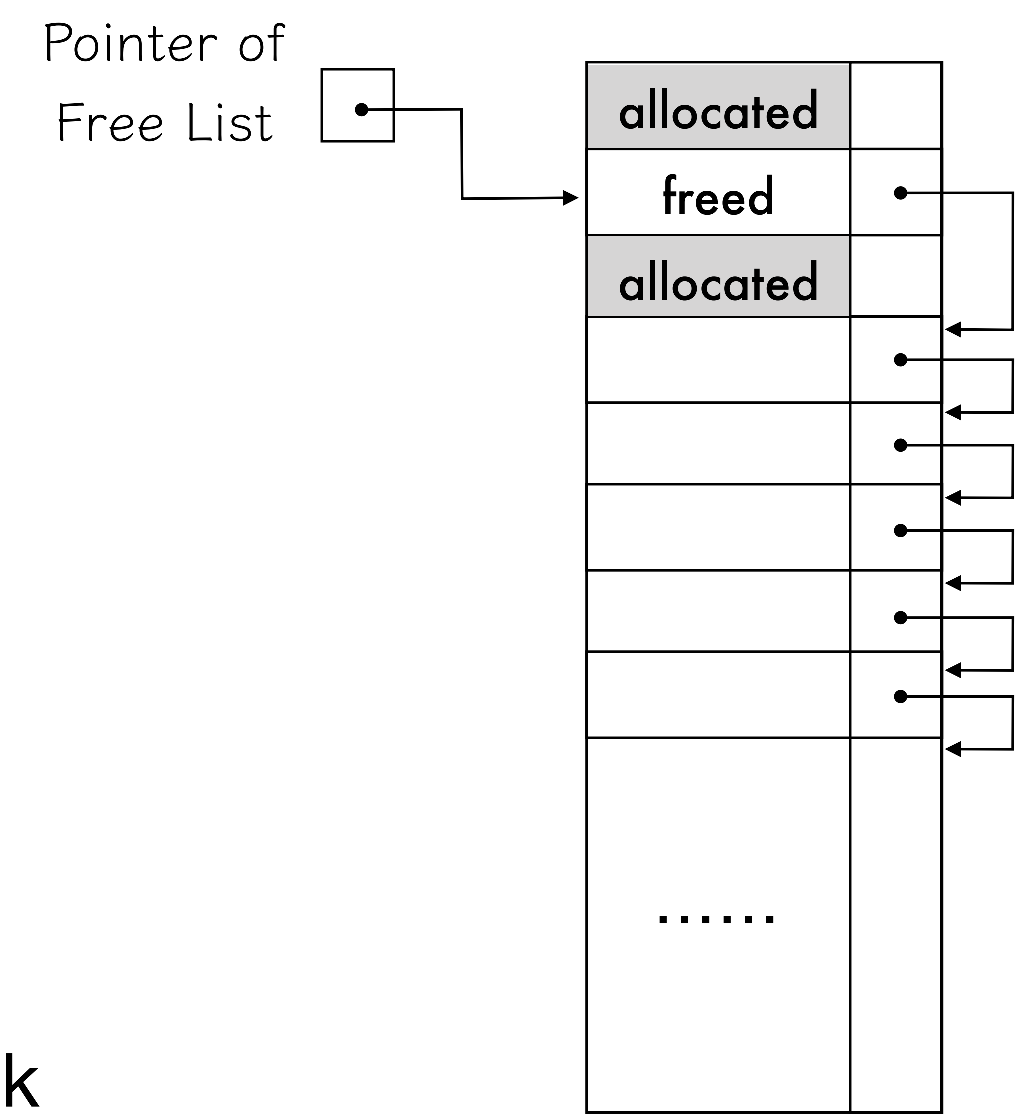
Fixed Length Blocks

- Divide the heap to multiple fixed length blocks.
- Using a free list to maintain the list of free blocks.
- When there is a request, the first block of the free list is assigned and the block is removed from the free list.
- When a block is freed (or deallocated), the block is back to the free list.



Fixed Length Blocks

- Divide the heap to multiple fixed length blocks.
- Using a free list to maintain the list of free blocks.
- When there is a request, the first block of the free list is assigned and the block is removed from the free list.
- When a block is freed (or deallocated), the block is back to the free list.

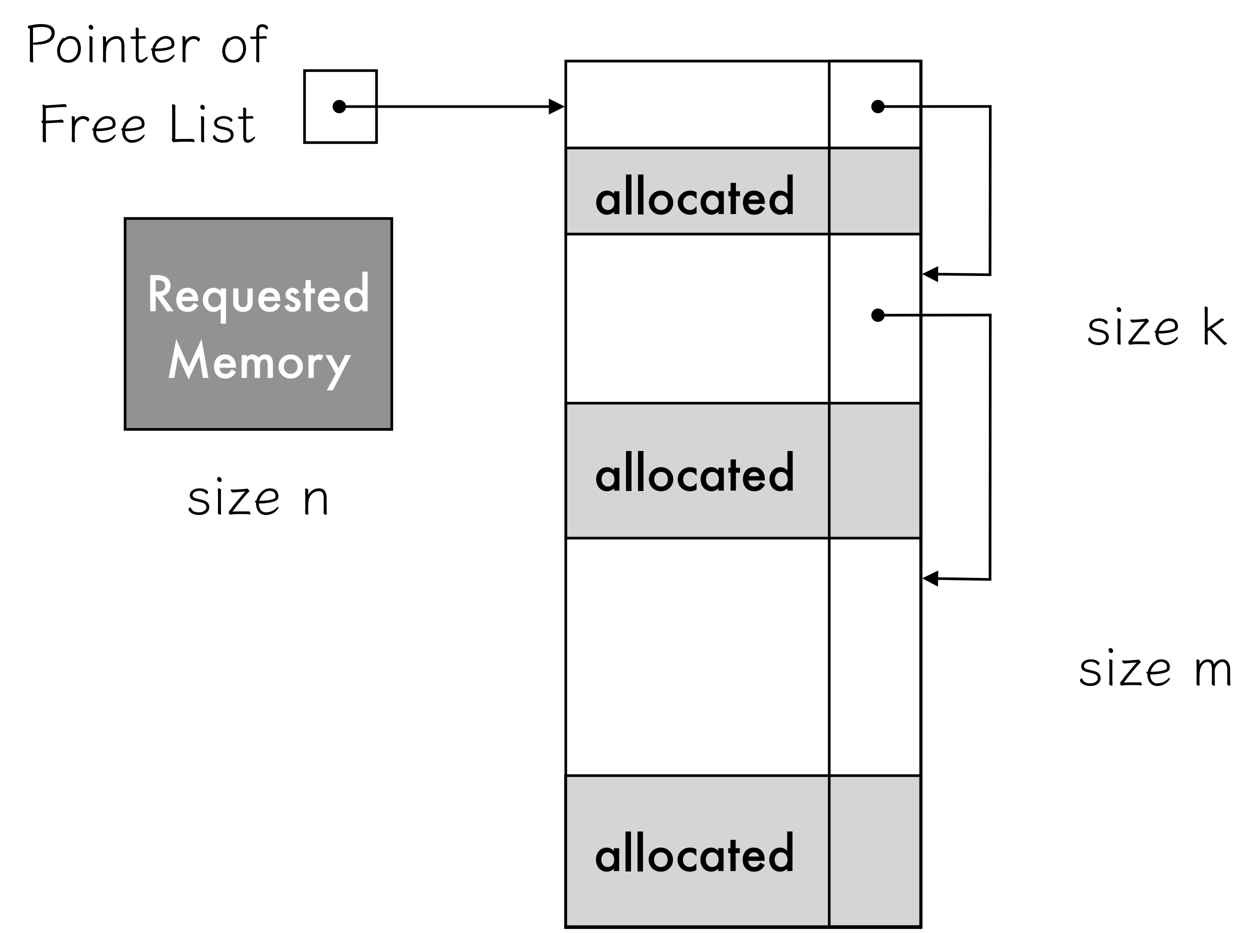


Variable Length Blocks

- Similar to fixed length blocks, it maintains a free list for available blocks.

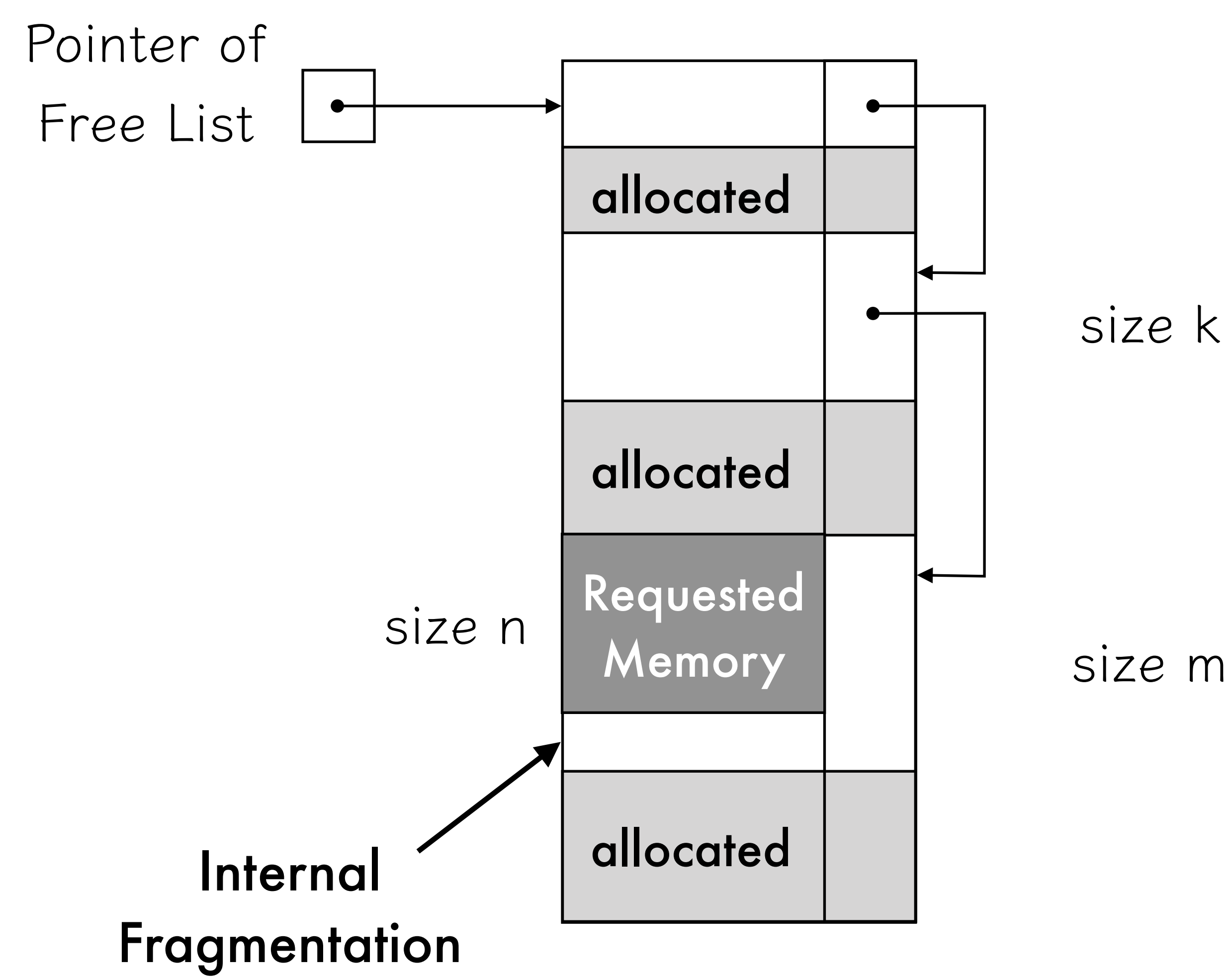
- The size of blocks can be different.

- When a request for memory of size n, it allocates a free block large enough to satisfy the request.



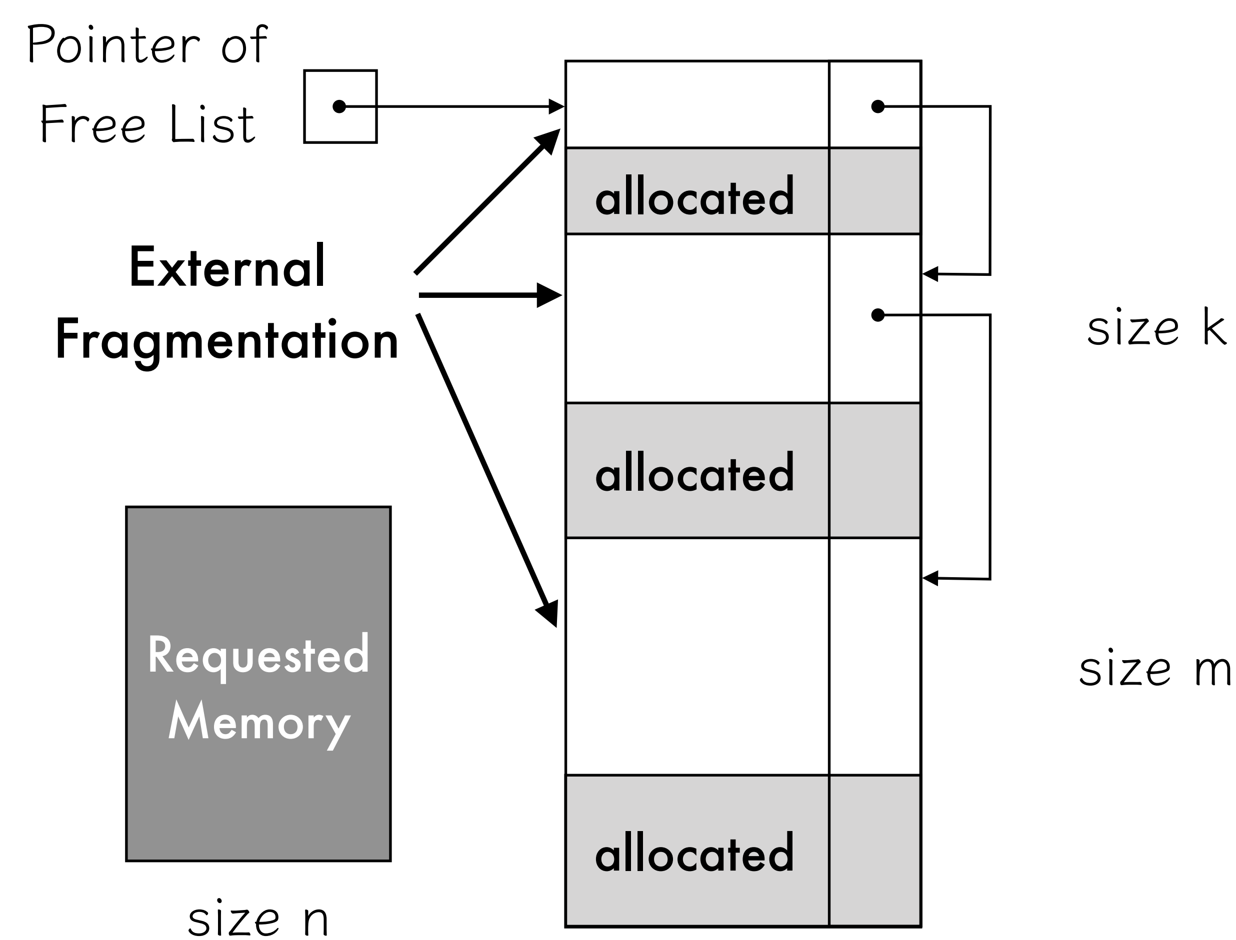
Fragmentation

- Variable length method causes fragmentation.
- Due to fragmentation, memory space is wasted, or it reduces the performance of programs.
- ***Internal Fragmentation***: Allocated block size is greater than the requested size.
    - m > n, then d = m - n is wasted.



Fragmentation

- ***External Fragmentation***
  - Due to the scattered free blocks, requested memory cannot be allocated,
  - even it there exists enough space.
  - m + k > n, but they are not consecutive.

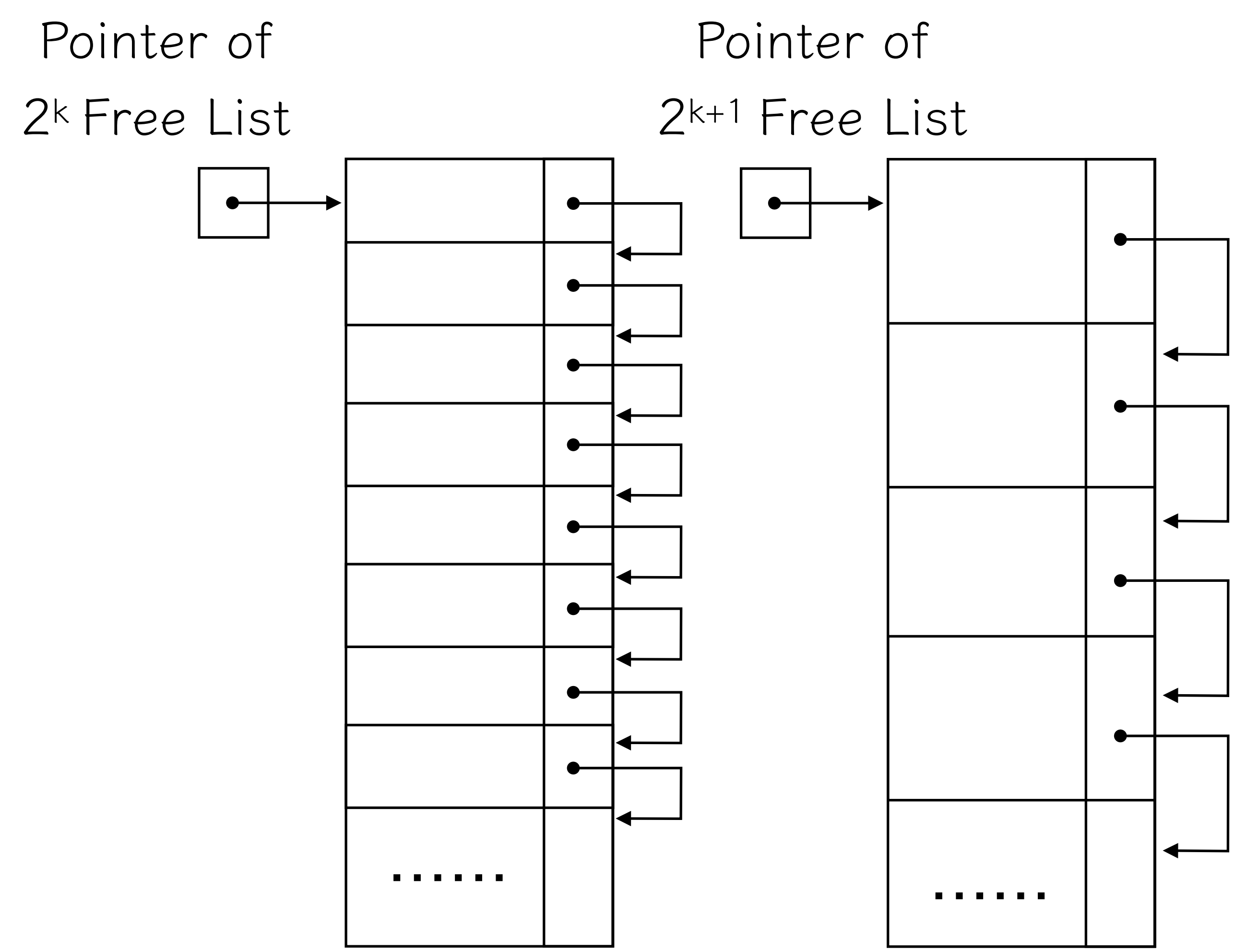


Using Single Free List

- When there is a request for memory allocation of size n,
- Directly use the free list.
  - First Fit: allocate the first block bigger than size n.
  - Best Fit: allocate the size $k >= n$ block which has the minimum $d = k - n$.
- Free Memory Compaction.
  - When the end of the heap is reached, move all active blocks to the end.

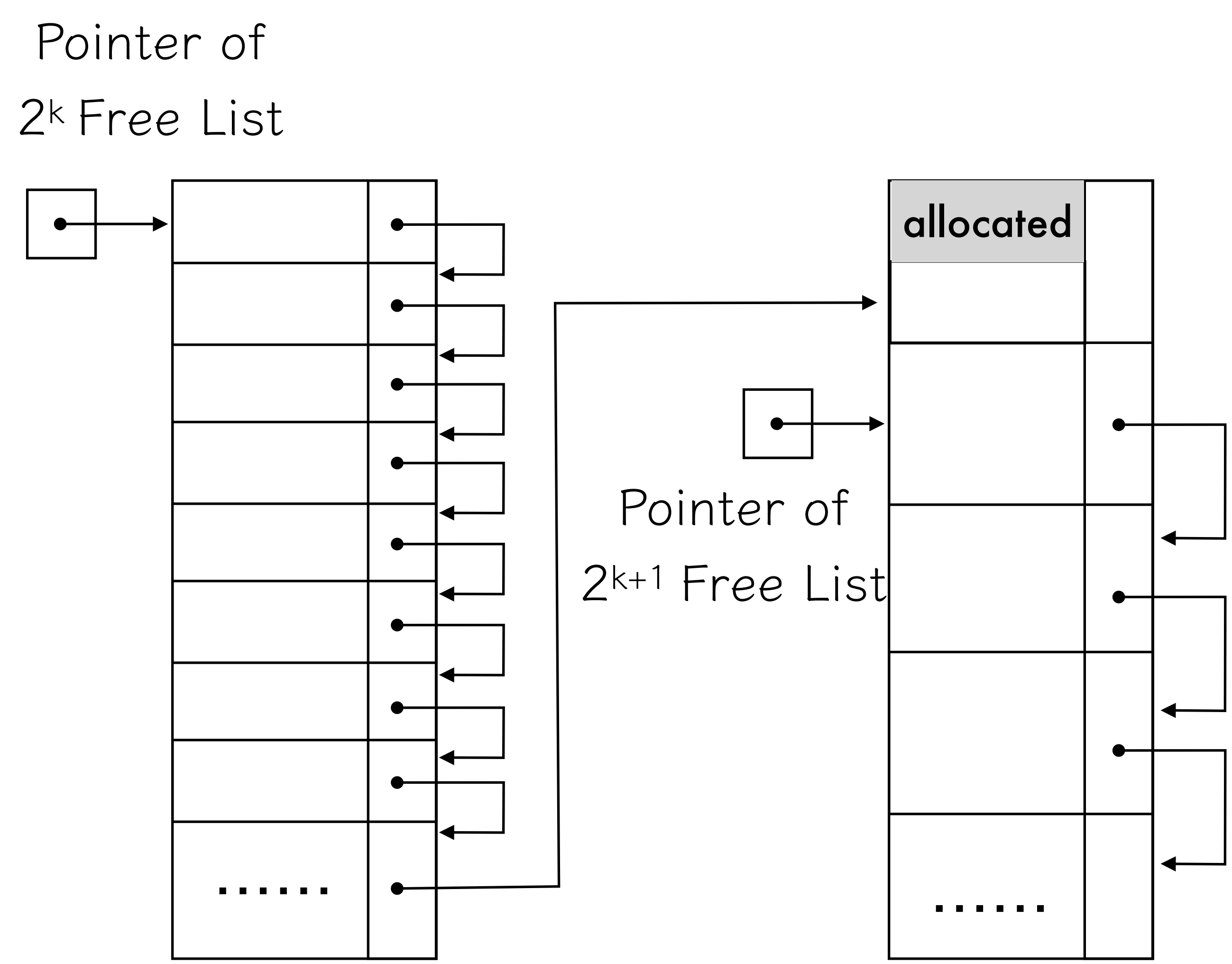
Multiple Free Lists

- **Buddy System**
  - Have multiple free lists with size power of 2 (i.e. 2^x).
  - For size n request, find a block from the free list of 2^k >= n blocks.
  - If there is no available block, then search 2^{k+1} free list next.



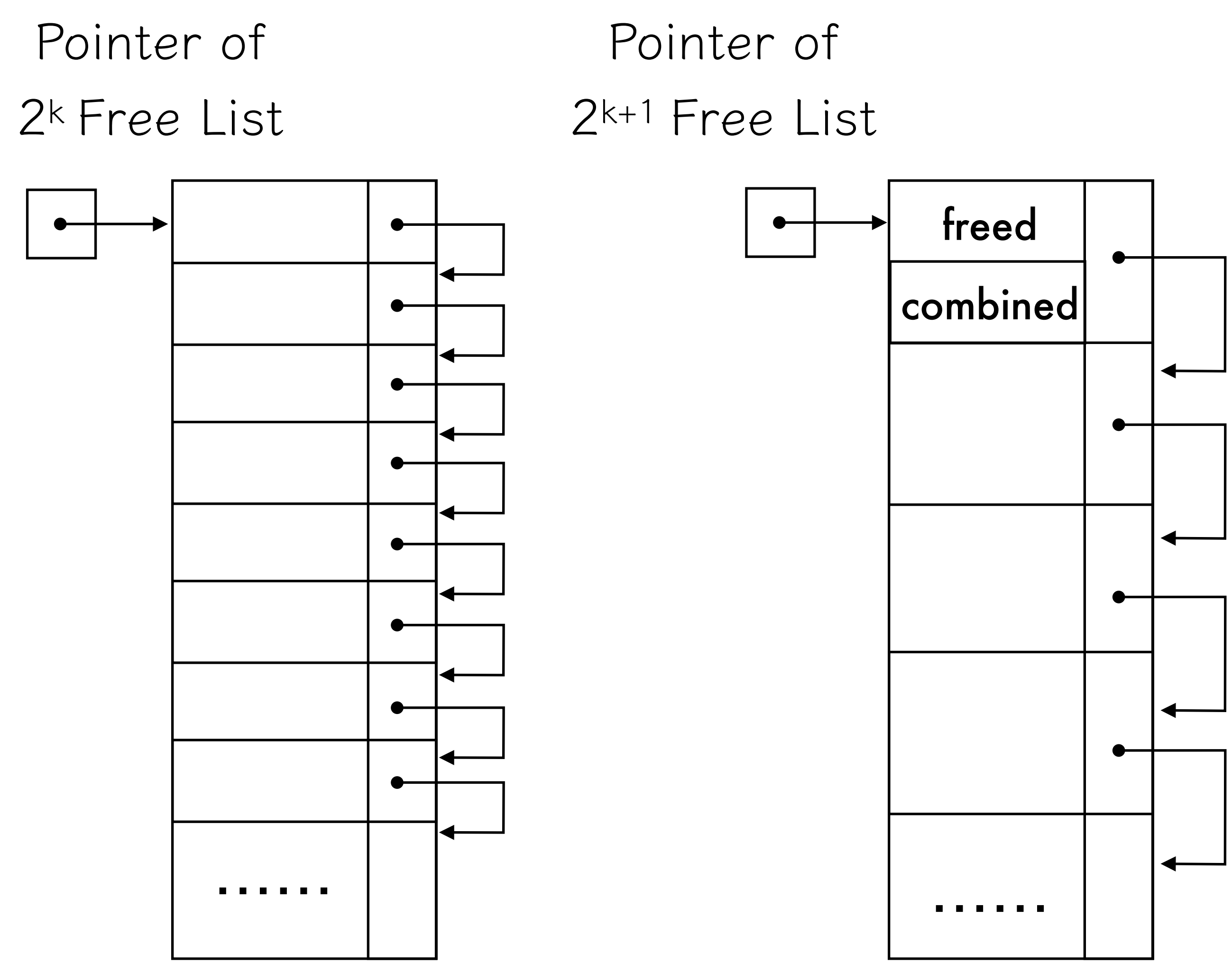
Multiple Free Lists

- When a free block is found in 2^{k+1} free list,
  - Split this block into two 2^k blocks.
  - Allocate one of them, and connect the other to 2^k free list.



Multiple Free Lists

- Next time the allocated block is freed,
    - Find its *buddy* which is resulted by the split, and check it is also free.
    - Combine them and attach it to 2^{k+1} free list again.



Control Flow and Recursion

Unbounded vs. Bounded

- We cannot see many "pure" bounded iteration statements.
- For bounded iteration, at the start of iterations, we can know ***the number of iteration***.
- If we can affect the loop conditions within in the loop's body, we cannot compute the number of iteration when starting.
- e.g.) In C, for statement is not pure bounded iteration.
  - ```
    for (int i=0; i<n; i++) {
            ...
            n += 1;      ← updating the condition is
                           possible in the body of the loop.
    }
    ```
    ```

Tail Recursion

- Recursion would be much more efficient, if we share the activation records for each recursive call.
- ***Tail recursion***: returns a return value without any additional computations.
- We may introduce a new variable to store intermediate results as parameters.

Recursion

```
int fact(int n) {
    if(n == 1)
        return 1;
    else
        return n*fact(n-1);
}
```

Tail Recursion

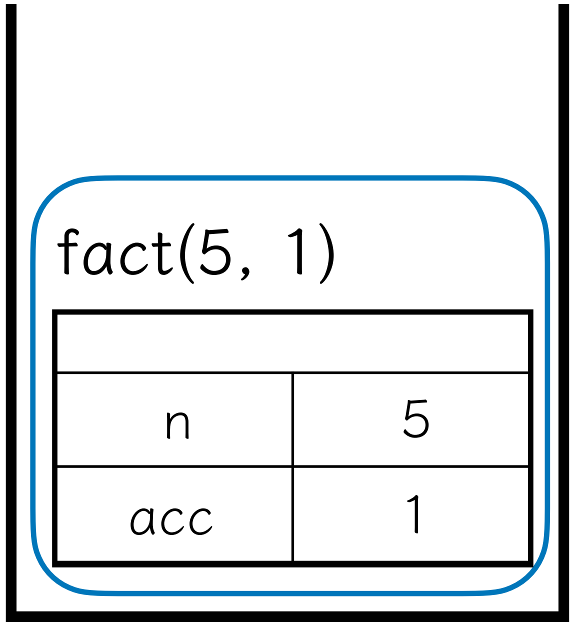
```
int fact(int n, int acc) {
    if(n == 1)
        return acc;
    else
        return fact(n-1, n*acc);
}
```

Directly returns parameters or return values

Tail Recursion

- Every time fact() is called at the last line, the activation record is overridden,
  - Instead of adding a new activation record.
- It is only possible because it does not need to wait for the return value of the recursive call.
  - On the other hand, `n*fact(n-1)` cannot be computed until `fact(n-1)` execution is completed.
  - So activation record requires a space to store the return value.

```
int fact(int n, int acc) {
    if(n == 1)
        return acc;
    else
        return fact(n-1, n*acc);
}
```

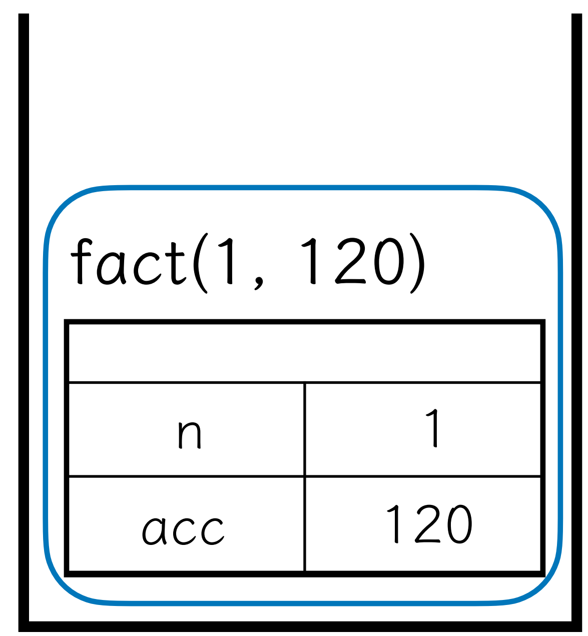


Single Activation Record

Tail Recursion

- Every time fact() is called at the last line, the activation record is overridden,
    - Instead of adding a new activation record.
- It is only possible because it does not need to wait for the return value of the recursive call.
    - On the other hand, `n*fact(n-1)` cannot be computed until `fact(n-1)` execution is completed.
    - So activation record requires a space to store the return value.

```
int fact(int n, int acc) {
    if(n == 1)
        return acc;
    else
        return fact(n-1, n*acc);
}
```



Single Activation Record

Euclidean Algorithm

- It is an algorithm to find the Greatest Common Divisor (GCD) of two numbers *a* and b.
- `gcd(a, b) = gcd(10, 6) = 2.`
- If `a % b = r and a > b, then gcd(a, b) = gcd(b, r).`

```
a = 2712, b = 888
gcd(a, b) = ?

2712 = 888*3 + 48
888 = 48*18 + 24
48 = 24*2 + 0

Recursive Calls

gcd(2712, 888)
→ gcd(888, 48)
→ gcd(48, 24)
```

Summary

- Dynamic Memory Management
  - Using Stack
  - Using Heap
- Iteration
- Recursion