Matej Dedina

A C Data Structure Book

BOOK I

February 14, 2026

Springer Nature

Preface

This book describes the process of designing and implementing a personal data structure library in the C programming language. Each implementation is centered around two types of implementations - an infinitely expandable structure able to hold any amount of elements, and a finite one able to hold only a certain maximum amount.

The goal of this work is not to be the go-to data structure book for C, on the contrary, the name is meant to represent an affordable book for the public which doesn't take itself seriously.

The book will be divided into parts classifying all structures into categories of chapters. Each chapter will explain the process of creating said structure and the code used to make it.

Since the code is in C a decent understanding of the programming language is required, that primarily includes - structures (struct), pointers, memory allocation, and last but certainly not least - function pointers.

All the source code will be available at github under the `Unlicense License`.

If I have to put a disclaimer I just want to clarify that the reason why some paragraphs may be written like this is because the book was created using LaTeXin Visual Studio Code and I hate seeing `Overfull \hbox` highlights.

Contents

Part I
Restricted sequential data structures

The name of the first part comes from the inability to find a categoric name for only stacks, queues and deques. Online sources categorize those as linear dynamic data structures, the problem is... linked lists are also part of this category. I want to put list implementations into a separate book part, thus I hereby coin the term `Restricted sequential data structures` as a subcategory ob dynamic linear data structures. I know nobody will take this declaration seriously, but for the sake of this book please bear with it.

Chapter 1
Stacks

Lets imagine we have a stack of books. The normal way we can add books to the stack is by putting them at the top, if we put a book with the cover pointing upwards one also knows what book exactly is at the top. To remove a book we just grab the top one and put it somewhere else.

By making space for another stack of books and adding all the books from the first one, while the constrains of putting and removing still remain, we get a new stack, but with a specific property that, when compared with our initial one, makes us tilt our head (literally). The new stack is just like the last one, but with the books in opposite order. This little property allows us to invert the order of any linear ordering of elements without knowing how they're implemented, if adding and removing of elements is allowed.

Continuing with the two stack analogy, what if, instead of books we stored a game of chess. By adding these moves one-by-one 1.1: E2-E4, E7-E5, G1-F3, B8-C6, D2-D4, E5-D4, F3-D4, F8-C5, C2-C3, D8-F6, D4-C6, F6-F2; we get probably the funniest chess game in modern history [1]. And on top of that if we remove the two moves from the top and reset the pieces (F6-F2 becomes F2-F6), iteratively it is possible to return to the initial state of the game. And to top it all of, as probably every data structure book and article about stacks writes this mechanism is used by search engines when you want to return to the previous pages and back.

Stacks are also used to store function calls in programming languages like C to be able to know to what previous in memory place the program should go to after the top function returns.

1.1 Implementations of a Stack

Every computer science student learns that the two most common ways to implement a stack in C are:

[1] that can be watched via `https://www.youtube.com/watch?v=e91MOXLX7Jw`

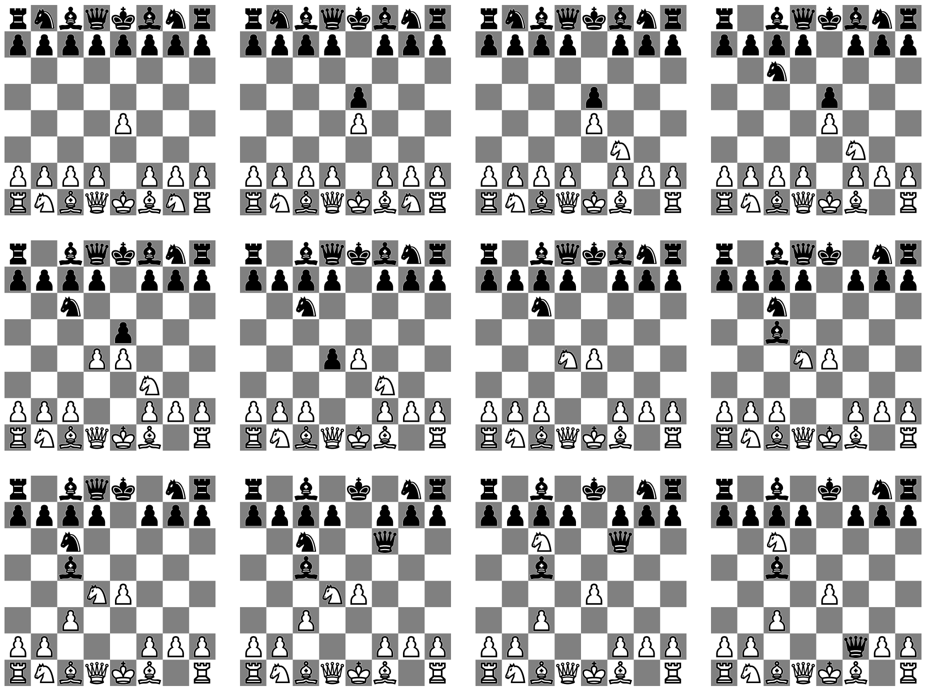

Fig. 1.1 The entire game of chess between content creators xQc and MoistCr1tikal with the latter coming on top.

- Singly linked lists
- Static/dynamic arrays

So why does this book have three subsections for stack implementations?

1.1.1 The humble linked list

For a more in-depth explanation of lists you can infer to PART II Linked lists, but in short a linked list is a linear data structure which allows us to store data sequentially as well as store a reference to the next data item 1.2.

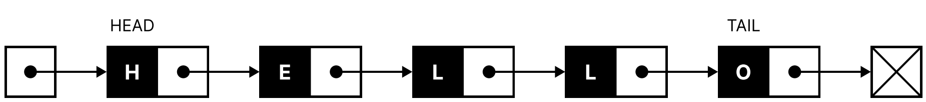

Fig. 1.2 A simple graphical linked list example.

The linked list consists of two special nodes which represent the beginning and the end - head and tail. Most linked list implementation will store the reference to the head, so it is generally easy to access the first element as oppose to the tail, where the user needs to follow all the arrows until they reach it.

Adding, removing and accessing the first top node only requires a reference to the head 1.3. Adding is just creating a new node, then making its reference point to the stack's head, and lastly we change the stack's head itself into the new node. Removal is storing the top node, then change stack's head to the next reference (head's next node), and destroying the removed node.

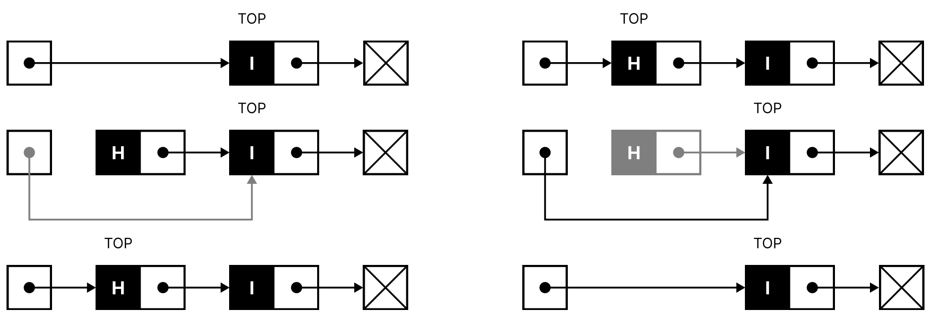

Fig. 1.3 Pushing element 'H' to the top of a linked list stack and creating 'HI' (left), and removing it (right).

1.1.2 The prideful array

The linked list is kinda complicated, and compared to our initial book analogy there's a better way to represent stacks - arrays. Another problem is cache locality, which makes linked list stacks slower since elements may be stored huge memory distances away and loading them to faster, but smaller, memory areas for speed may be impossible. On the other hand, there are some improvements that can be made even with lists, but back to the topic.

An array is just a linear data structure made up of continuous memory. This simple concept, however, makes it one of the most versatile data structures in all of programming.

Adding, removing and accessing the first top node only requires a numerical index value that can be inferred through the stack's length We just store the LENGTH of the stack, then add an element by putting it into the INDEX = LENGTH and incrementing LENGTH by one; removing an element requires decrementing LENGTH by one and making INDEX = LENGTH. Just don't forget to check if a stack's maximum length is less than maximum/capacity size during the push; and length not being zero while popping. Accessing the topmost element is just INDEX = LENGTH - 1, without the decrement 1.4.

The biggest concern is that arrays aren't infinite and if we want to add an arbitrary number of elements we need to expand it - if we have space then we simply increase the capacity value, one must also store a changing capacity values for dynamically growing arrays and a constant maximum for static ones; else if the array occupies the maximum available chunk of memory all elements must be moved to a bigger chunk; and when that fails, because no chunk exists, everything's screwed and we must terminate.

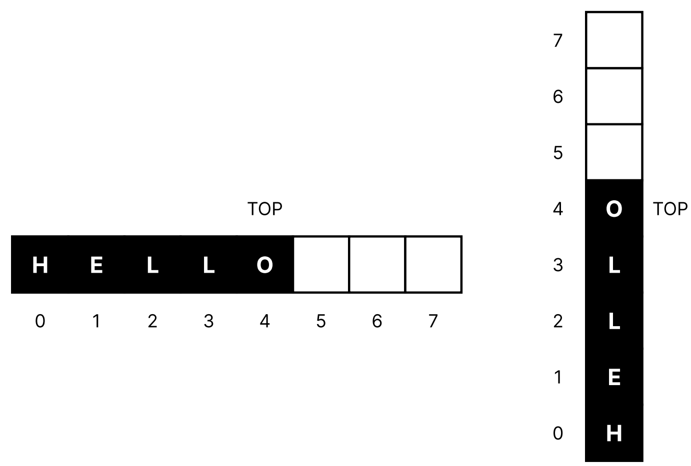

Fig. 1.4 A simple graphical array example. The top element is also the last element in the array.

1.1.3 Singly linked list of array elements

Combine the best of humbleness with pride and you get this. It's just a linked list, but the node is actually made up of an array of elements instead of just a singleton. A mathematical person may say that the previous implementations are but a simple subset of this superset of a stack structure, but i digress 1.5.

The main benefit is that the list-array works like a array when accessing elements; and like a list when shortening/expanding it (we don't need to resize an array, only add a new node to the top). Adding and removing elements just combines the array method until full/empty, where it then performs the list operations. The implementation is more like a compromise than the best of both worlds.

To get the top element we can store the total element count as LENGTH, then simply getting the head node gives us access to the array with the top element so that lastly we only need to do INDEX = LENGTH - 1 mod CHUNK, where CHUNK is the overall array size.

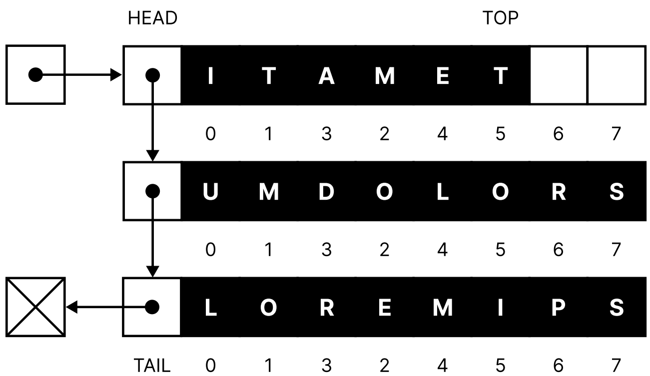

Fig. 1.5 Graphical example of singly linked list of arrays. The top element is the last element in the head node's array.

1.2 Array stack structure code

At the end I have decided to pick simple array-based stacks for one simple reason - continuous memory layout. I want to allow users to sort elements in the stack in order the have the same functionality as a priority queue, i.e. retrieving elements based on a priority, like the minimum. Of course the priority would be in reverse since the stack would get the last element in the array. It is possible to sort elements using the aforementioned implementations, but that requires creating a temporary array to copy elements back and forth. In chapter **??** the double ended queue data structure, which is implemented similar to a list of arrays, can also be used as a stack with the same list-array advantages.

1.2.1 The infinite stack

The `istack` code uses a dynamic array of elements based on user-defined size of single element, like using `sizeof`. The three members `size`, `length` and `capacity` represent single element size, current stack length and maximum capacity before resizing is required. Lastly, the memory allocator allocates, reallocates and frees the elements array and other temporary structures used in certain functionalities via Listing 1.2.1.

```
/// @brief Stack data structure.
typedef struct infinite_stack {
    char * elements;              // array of elements
    size_t size, length, capacity; // size of single element,
        structure length and its capacity
    memory_s const * allocator;
} istack_s;
```

1.2.2 Creating the stack

The `create_istack` function creates an empty infinite stack structure storing the element size and allocator.

```
/// @brief Creates an empty structure.
/// @param size Size of a single element.
/// @return Stack structure.
istack_s create_istack(size_t const size);
```

The creation process uses a compound literal to quickly create and return a new infinite stack structure. It only needs to set the single element `size` and `allocator` members.

```
{
    ...
    return (istack_s) { .size = size, .allocator = &standard };
}
```

1.2.3 The makings of a stack

The `make_fstack` function, similar to `create_fstack`, creates an infinite stack structure, but with the ability to specify a custom memory allocator.

```
/// @brief Creates an empty structure.
/// @param size Size of a single element.
/// @param allocator Custom allocator structure.
/// @return Stack structure.
istack_s make_istack(size_t const size, memory_s const * const
    allocator);
```

The code is basically the same as the aforementioned function, but with the `standard` allocator replaced with the parameter one.

```
{
    ...
    return (istack_s) { .size = size, .allocator = allocator };
}
```

1.2.4 Stack destruction

Next, `destroy_istack` destroys the stack and all its elements while at the same time making it invalid for future usage. The parameters include a pointer to the stack and a function pointer taking in each element, allowing the user to manipulate with elements prior to their destruction. The `destroy` function can primarily be used to free memory, like strings or character arrays, from the heap.

```c
/// @brief Destroys a structure, and its elements and makes it
    unusable.
/// @param stack Structure to destroy.
/// @param destroy Function pointer to destroy a single element.
void destroy_istack(istack_s * const stack, set_fn const destroy);
```

Since our elements are stored in an array, i.e. continuous memory, it is possible to write a for loop with a pointer representing a single element to call the `destroy()` function pointer on. The variable then gets incremented by the specified size until the last valid memory address is reached.

```c
{
    ...
    // iterate over each element and call destroy function on it
    for (char * e = stack->elements; e < stack->elements + (stack->
        length * stack->size); e += stack->size) {
        destroy(e);
    }
    ...
```

After destroying each element the next step is to free the element's array using the specified `allocator` struct member.

```c
    ...
    stack->allocator->free(stack->elements, stack->allocator->
        arguments);
    ...
```

All that's left is to invalidate the stack, or set everything to zero, via calling `memset` from `<string.h>`.

```c
    ...
    memset(stack, 0, sizeof(istack_s));
}
```

The `memset` function allows us to effectively set every member of a struct or any array to zero. It should only be used to set values to either all zero or all one (by using `-1` as the second parameter).

Listing 1.1 The `memset` function prototype.

```c
#include <string.h>

void *memset(void s[n], int c, size_t n);
```

> The `memset()` function fills the first n bytes of the memory area pointed to by s with the constant byte c.

1.2.5 Clearing up the stack

Similar to `destroy_istack`, `clear_istack` can be used to destroy each element in our structure, but the stack remains valid for next use. This is achieved by freeing the elements array and setting the stack's length and capacity to zero, and elements array NULL.

```
/// @brief Clears a structure, and destroys its elements, but
    remains usable.
/// @param stack Structure to destroy.
/// @param destroy Function pointer to destroy a single element.
void clear_istack(istack_s * const stack, set_fn const destroy);

{
    ...
    // set lenght to zero
    stack->length = stack->capacity = 0;
    stack->elements = NULL;
}
```

1.2.6 The deep/shallow copy

Creating a copy, either with all nested sub-elements recreated deeply or referenced shallowly, can be done with the `copy_istack` function. The main idea is to use a special `copy` function pointer which takes in a destination element memory address of the replica structure and a source address from the parameter and copies source element into destination element and returning the destination address. This functionality is similar to the `memcpy` function in `<string.h>`, which takes a destination address, source address, and the size of the memory area to copy.

Listing 1.2 The `memcpy` function prototype from the Linux/UNIX manual page.

```
#include <string.h>

void *memcpy(void dest[restrict n], const void src[restrict n],
    size_t n);
```

Linux manual page - memcpy(3)

The `memcpy()` function copies n bytes from memory area `src` to memory area `dest`. The memory areas must not overlap. Use `memmove(3)` if the memory areas do overlap.

```
/// @brief Creates a copy of a structure and all its elements.
/// @param stack Structure to copy.
/// @param copy Function pointer to create a deep/shallow copy of a
     single element.
/// @return Stack structure.
istack_s copy_istack(istack_s const * const stack, copy_fn const
    copy);
```

First and foremost, a replica stack structure is created and initialized, with all members except the elements array being directly copied. Then the original `allocator` is used to generate a memory array.

```
{
    // create replica to initialize and return
    istack_s const replica = {
        .capacity = stack->capacity, .length = 0, .size = stack->
            size,
        .elements = stack->allocator->alloc(stack->capacity * stack
            ->size, stack->allocator->arguments),
    };
```

All elements are later copied individually from source stack's memory at index position into replica's elements array.

```
    // initialize replica's elements array with stack's elements
    for (size_t i = 0; i < stack->length; ++i) {
        size_t const offset = i * stack->size;
        copy(replica.elements + offset, stack->elements + offset);
    }

    return replica;
}
```

1.2.7 Checking emptiness

Since we can't remove elements from an empty stack, the function available to help with this is the `is_empty_istack`.

```
/// @brief Checks if structure is empty.
/// @param stack Structure to check.
/// @return 'true' if empty, 'false' if not.
bool is_empty_istack(istack_s const * const stack);

{
    ...
    return !(stack->length); // return negated length
}
```

1.2.8 Pushups to the top

To add an element one can use the the `push_istack` function. Since the stack is implemented using a resizable array checking if maximum capacity has been reached is a must. The index of where to push the element can be gained through the stack length member.

```
/// @brief Pushes a single element to the top of the structure.
/// @param stack Structure to push into.
/// @param element Element buffer to push.
void push_istack(istack_s * const restrict stack, void const *
    const restrict element);
```

Firstly, checking ig stack's length has reached the capacity allows for the resizing into a new chunk of memory. Since the structures grow exponentially starting at a specific power of two chunk it is important to resize it into `ISTACK_CHUNK` if `length` member is zero and else to continue with doubling.

```
{
    ...
    if (stack->length == stack->capacity) { // if length is equal
        to capacity the array must expand linearly
        size_t const capacity = stack->length ? stack->length *
            CERPEC_FACTOR : ISTACK_CHUNK;
        _istack_resize(stack, capacity);
    }
```

As the size of a single element is user-define, the only optimal way to copy the element into our structure is to use `memcpy`. Because the `element` parameter and `elements` stack member are stored in two different memory locations `memcpy` and its speed can be utilized.

```
    ...
    // push element knowing the elements array can fit it
    memcpy(stack->elements + (stack->length * stack->size), element
        , stack->size);
    stack->length++;
}
```

1.2.9 Popping it

Removal of elements is done through the `pop_istack` function. If the structure is empty the program logically errors.

```
/// @brief Pops a single element from the top of the structure.
/// @param stack Structure to pop from.
/// @param buffer Element buffer to save pop.
void pop_istack(istack_s * const restrict stack, void * const
    restrict buffer);
```

Popping the top element is as simple as getting the element at decrementing `lenght`, using it as the index position and copying into a temporary variable parameter.

```
{
    ...
    // remove element from elements array
    stack->length--;
    memcpy(buffer, stack->elements + (stack->length * stack->size),
         stack->size);
```

The structure adjusts itself during growth, therefore it is important to not waste memory. Thus after reaching a smaller power of two capacity the stack automatically shrinks. Chunk length is a special case where if the capacity goes below it, the memory array stays the same, but only until reaching zero. After that it is set to zero.

```
    ...
    if (stack->length <= stack->capacity / CERPEC_FACTOR && (stack
        ->length > ISTACK_CHUNK || !stack->length)) {
        _istack_resize(stack, stack->length);
    }
}
```

1.2.10 Reaching the peep

Peeping the last added element via `peep_istack` is a little bit like popping it.

```
/// @brief Peeps a single element from the top of the structure.
/// @param stack Structure to peep.
/// @param buffer Element buffer to save peep.
void peep_istack(istack_s const * const restrict stack, void *
    const restrict buffer);
```

The idea is to just get the element index at `length - 1` and save it into a temporary parameter.

```
{
    ...
    // only copy the top element into the buffer
    memcpy(buffer, stack->elements + ((stack->length - 1) * stack->
        size), stack->size);
}
```

1.2.11 Each element

Each element can be accessed and manipulated through the special `each_istack` function which iterates beginning not at the top, but at the bottom, and them making its way up to the top. The reason it starts at the bottom is because it make printing the values easier.

```
/// @brief Iterates over each element in structure starting from
    the beginning.
/// @param stack Structure to iterate over.
/// @param handle Function pointer to handle each element reference
     using generic arguments.
/// @param arguments Generic arguments to use in function pointer.
void each_istack(istack_s const * const restrict stack, handle_fn
    const handle, void * const restrict arguments);
```

All the element are in an array, thus iterating over them is as simple as using an empty for loop. The `handle` function pointer, together with `arguments`, allows direct access to the element, thus manipulation is possible. If `handle` returns false, the iteration automatically stops, kinda like the `break` keyword.

```
{
    ...
    // iterate over each element from bottom to the top of stack
    for (char * e = stack->elements; e < stack->elements + (stack->
        length * stack->size) && handle(e, arguments); e += stack->
        size) {}
}
```

1.2.12 Applying mostly sorting

The `apply_istack` gives access to all the elements as if they were inside a continuous array, which doesn't really matter for stacks, but it is extremely useful for structures in other chapters, like **??**. The main use of `apply_istack` is to sort the array of element as fast as the sorting function pointer `process` and its generic `arguments` allow.

```
/// @brief Apply each element in structure into an array to manage.
/// @param stack Structure to map.
/// @param process Function pointer to process array of elements
    using structure length and arguments.
/// @param arguments Generic arguments to use in function pointer.
void apply_istack(istack_s const * const restrict stack, process_fn
     const process, void * const restrict arguments);
```

Similarly to the `each_istack`, the `apply_istack` only has one line of code - the `process` function pointer itself.

```
{
    ...
    // process stack elements as an array (as a whole)
    process(stack->elements, stack->length, arguments);
}
```

Part II
Linked lists

Appendix A
Chapter Heading

Use the template *appendix.tex* together with the Springer Nature document class `SNmono` (monograph-type books) or `SNmult` (edited books) to style the appendix of your book.

A.1 Section Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations.

A.1.1 Subsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. A.1.

For multiline equations we recommend to use the `align` environment.

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= \mathbf{c} \\ \mathbf{a} \times \mathbf{b} &= \mathbf{c} \qquad \text{(A.1)} \end{aligned}$$

A.1.1.1 Subsubsection Heading

Instead of simply listing headings of different levels we recommend to let every heading be followed by at least a short passage of text. Furtheron please use the LaTeX automatism for all your cross-references and citations as has already been described in Sect. A.1.1.

Please note that the first line of text that follows a heading is not indented, whereas the first lines of all subsequent paragraphs are.

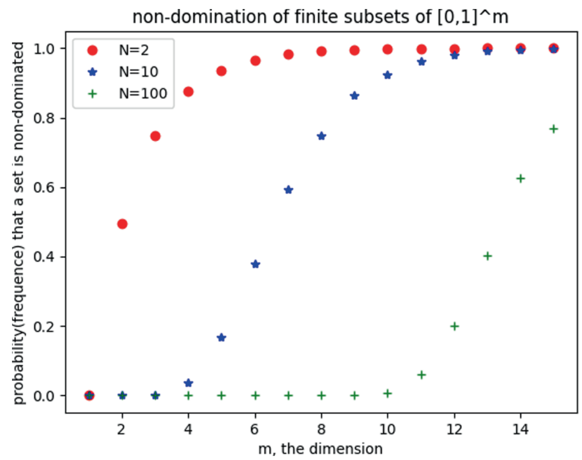

Fig. A.1 Please write your figure caption here

Table A.1 Please write your table caption here

Classes	Subclass	Length	Action Mechanism
Translation	mRNA[a]	22 (19–25)	Translation repression, mRNA cleavage
Translation	mRNA cleavage	21	mRNA cleavage
Translation	mRNA	21–22	mRNA cleavage
Translation	mRNA	24–26	Histone and DNA Modification

[a] Table foot note (with superscript)

Glossary

Use the template *glossary.tex* together with the Springer Nature document class SV-Mono (monograph-type books) or SVMult (edited books) to style your glossary in the Springer Nature layout.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

glossary term Write here the description of the glossary term. Write here the description of the glossary term. Write here the description of the glossary term.

Solutions

Problems of Chapter ??

?? The solution is revealed here.

?? **Problem Heading**
(a) The solution of first part is revealed here.
(b) The solution of second part is revealed here.

Index