# Data Structures. Data Structures in C.

Pepe Gallardo, 2024.

Dpto. Lenguajes y Ciencias de la Computación.

University of Málaga

## Modules in C

- A non-trivial program is usually divided into several modules.
- Each module is a separate source file containing a set of related functions.
- Modules are compiled separately and then linked together.
- Modules are usually defined in header files (.h) and implemented in source files (.c).
- Header files contain the declarations of the public functions, types and global variables provided by the module.
- Source files contain the implementation of the functions, types and variables declared in the header file and any other private declarations needed for the implementation.

# Linked Structures

# Linked Structures

- **Definition**: A linked structure is a data structure where elements are stored in nodes.
- **Node Composition**: Each node contains a value (element) and a pointer to the next node in the linked structure.
- **End of structure**: The last node has a `NULL` pointer, indicating the end of the structure.
- **Access**: The structure is represented by a pointer to the first node. Starting from this pointer, we can access all nodes by following the pointers.

## Advantages:

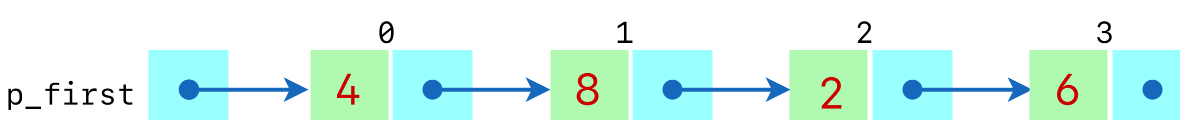
- **Dynamic Size**: The structure can grow and shrink as needed.
- **Efficient Insertions/Deletions**: No need to move elements around.

## Disadvantages:

- **Slower Access**: Accessing arbitrary elements is slower compared to arrays (O(n) vs O(1)).
- **Memory Overhead**: Each node requires additional memory for the pointer to the next node.

## Example:

- The following picture shows a linked structure of integers:

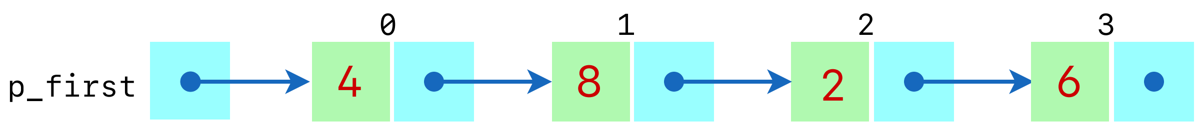


# Definition of a Linked Structure in C

- The following C definition is used to represent a linked structure of integers:
  - `struct Node` represents a node in the linked structure. Each node contains:
    - An integer element.
    - A pointer to the next node in the linked structure.

```
struct Node {
  int element;
  struct Node* p_next;
};
```

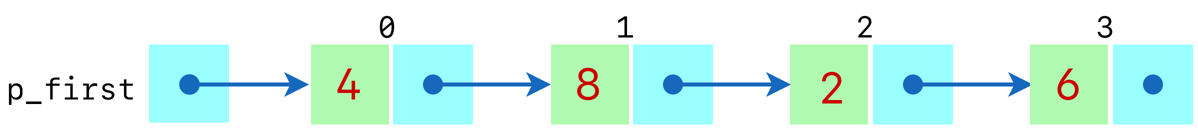
- Integer `element` in the struct is represented in green/red and `p_next` pointer in blue in the following picture:



## Representation of an Empty Linked Structure

- An empty linked structure is represented by a `NULL` pointer, as the first node does not exist:


- A non-empty linked structure is represented by a pointer to its first node. Notice that a `NULL` pointer is also used as a "end of structure" marker in the `p_next` component of the last node:



# The `LinkedStructure.h` Header File

- Notice that functions that modify the linked structure receive a pointer to the pointer to the first node. This is necessary because the pointer to the first node may change when the structure is modified.

```c
#ifndef LINKEDSTRUCTURE_H // Conditional inclusion
#define LINKEDSTRUCTURE_H // Avoids multiple inclusion

#include <stdbool.h>
#include <stddef.h>

// A Node in the linked structure
struct Node {
  int element;
  struct Node* p_next;
};

struct Node* LinkedStructure_new();
struct Node* LinkedStructure_copyOf(const struct Node* p_first);
bool LinkedStructure_isEmpty(const struct Node* p_first);
size_t LinkedStructure_size(const struct Node* p_first);
void LinkedStructure_clear(struct Node** p_p_first);
void LinkedStructure_append(struct Node** p_p_first, int element);
void LinkedStructure_prepend(struct Node** p_p_first, int element);
void LinkedStructure_insert(struct Node** p_p_first, size_t index, int element);
int LinkedStructure_get(const struct Node* p_first, size_t index);
void LinkedStructure_set(struct Node* p_first, size_t index, int element);
void LinkedStructure_delete(struct Node** p_p_first, size_t index);
void LinkedStructure_print(const struct Node* p_first);
#endif
```

# The `LinkedStructure.c` Source File

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "LinkedStructure.h"

/// @brief Constructor for a new Node
/// @param element Element to be stored in node
/// @param p_next Pointer to next to be stored in node
/// @return Pointer to new node
static struct Node* Node_new(int element, struct Node* p_next) {
  struct Node* p_node = malloc(sizeof(struct Node));
  assert(p_node != NULL && "Node_new: not enough memory");

  p_node->element = element;
  p_node->p_next = p_next;
  return p_node;
}

/// @brief Destructor for a Node. Frees memory allocated for node and sets pointer to node to NULL
/// @param p_p_node Pointer to pointer to node to be freed
static void Node_free(struct Node** p_p_node) {
  free(*p_p_node);
  *p_p_node = NULL;
}

struct Node* LinkedStructure_new() {
  return NULL;
}

... // rest of the implementation ...
```

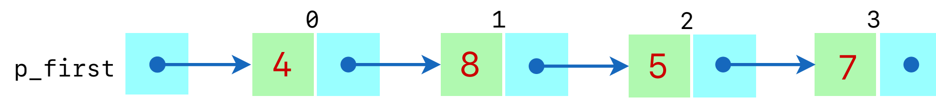
## Computing the Size of a Linked Structure

- The size of a linked structure is the number of nodes it contains.
- The size is computed by traversing the structure and counting the number of nodes.
- We start at the first node and traverse the structure by following the `p_next` pointers, until we reach the end of the structure (a node whose `p_next` pointer is `NULL`).
- We count the number of nodes visited during the traversal.

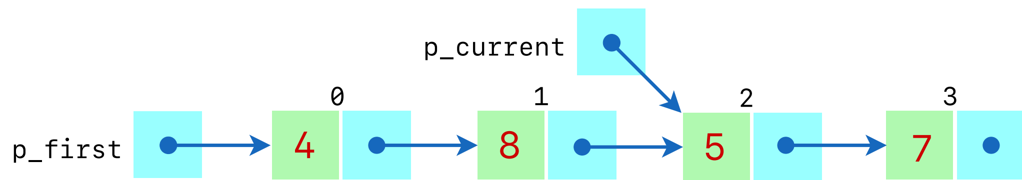
```
size_t LinkedStructure_size(const struct Node* p_first) {
  size_t size = 0;
  ... to complete
  return size;
}
```

# Getting the Element at a Given Position in a Linked Structure

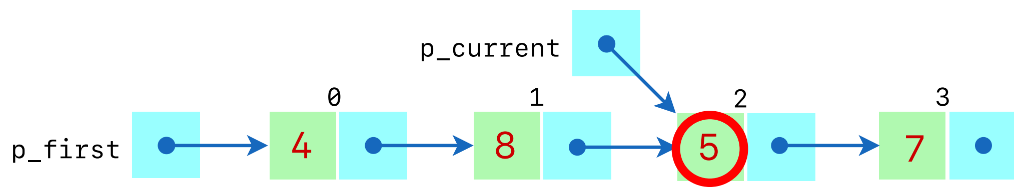
- We are going to `get` the element at position 2 in the following linked structure:



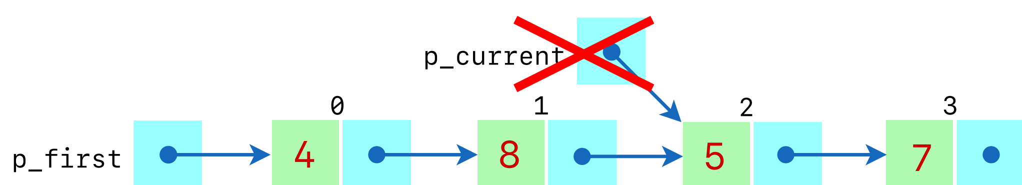

- Using a pointer (`p_current`), we start at the first node and traverse the structure until we reach the node at the desired position:




- We then return the element stored in that node:

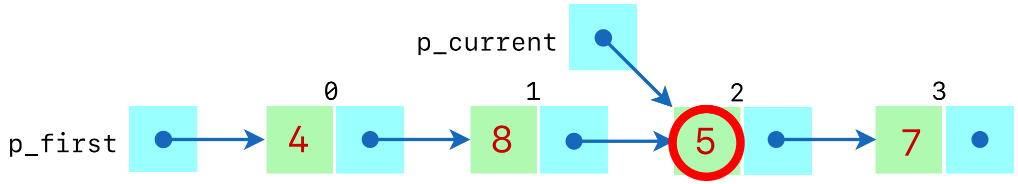



- The `p_current` pointer is an automatic variable so it is deallocated when the function returns:



# Setting the Element at a Given Position to a New Value in a Linked Structure

- The same algorithm as for `get` is used to reach the node at the desired position.
- We then update the element stored in that node through the pointer `p_current`:

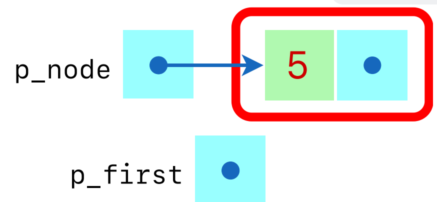



- For both `get` and `set` operations, it's important to handle **invalid positions** properly.

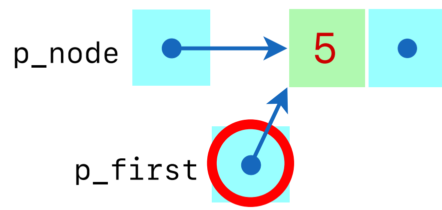
# Prependlng an Element at the Beginning of an Empty Linked Structure

- Starting from an empty linked structure, we are going to `prepend` element 5 at the beginning:

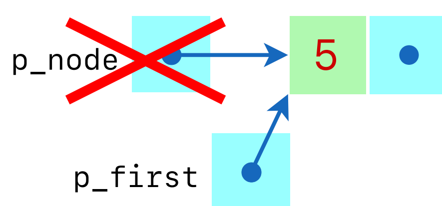

- A new node is allocated with element 5 and its `p_next` pointer is set to `NULL`:




- The pointer to the first node is updated to point to the new node:

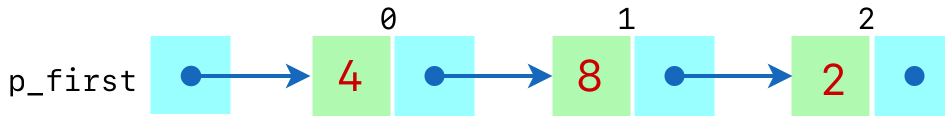



- `p_node` is an automatic variable so it is deallocated when the function returns:

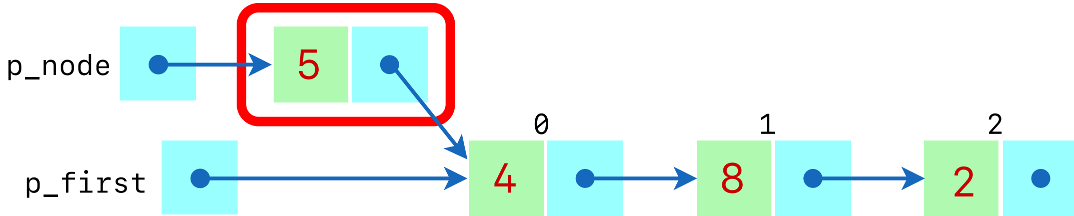


# Prepending an Element at the Beginning of a Non-Empty Linked Structure

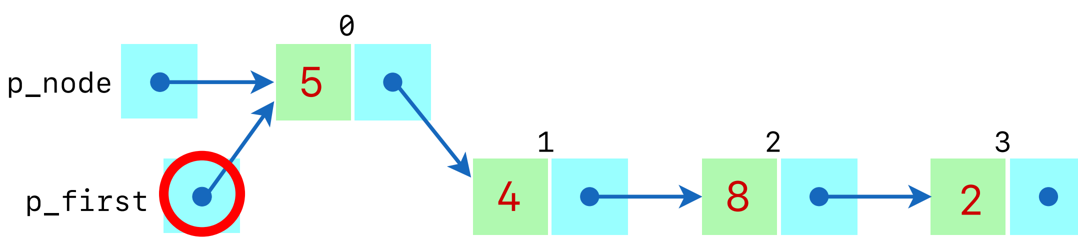
- Starting from this configuration, we are going to `prepend` element 5 at the beginning of this linked structure:



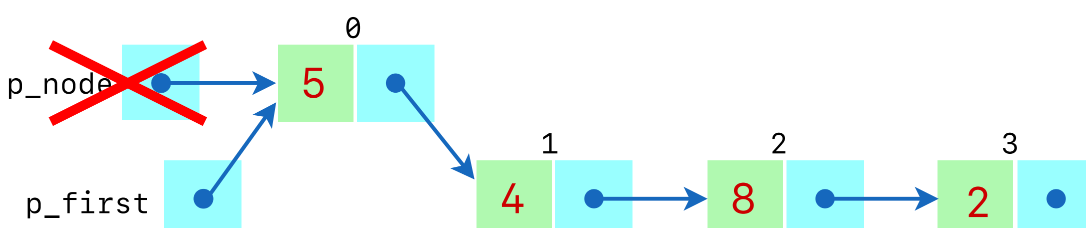

- A new node is allocated with element 5 and its `p_next` pointer is set to the current first node:




- The pointer to the first node is updated to point to the new node:



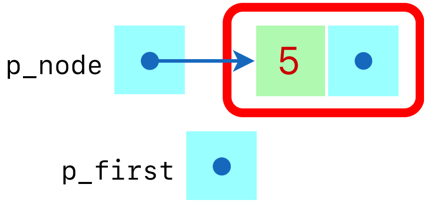

- `p_node` is an automatic variable so it is deallocated when the function returns:



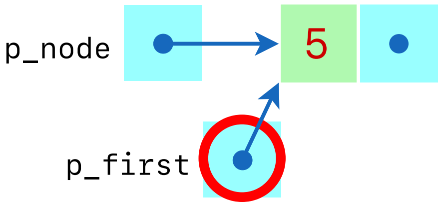
# Appending an Element at the End of an Empty Linked Structure

- Starting from an empty linked structure, we are going to `append` element 5 at the end:

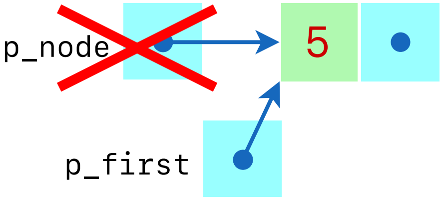

- A new node is allocated with element 5 and its `p_next` pointer is set to `NULL`:




- The pointer to the first node is updated to point to the new node:

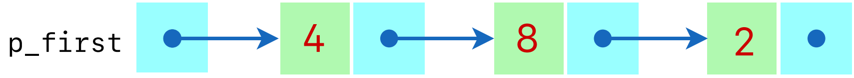



- `p_node` is an automatic variable so it is deallocated when the function returns:

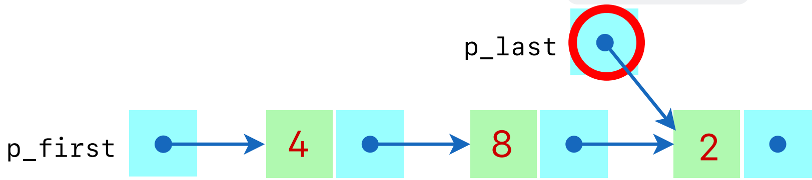


# Appending an Element at the End of a Non-Empty Linked Structure

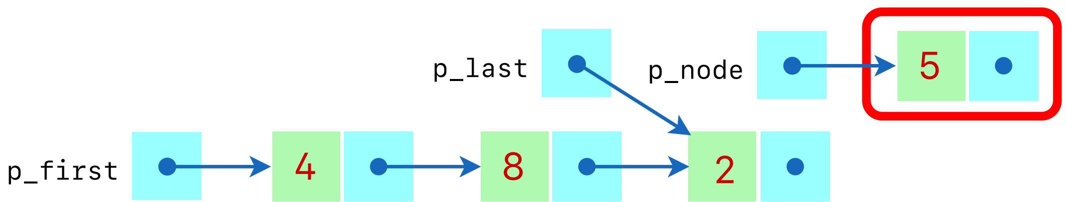
- Starting from this configuration, we are going to `append` element 5 at the end of this linked structure:



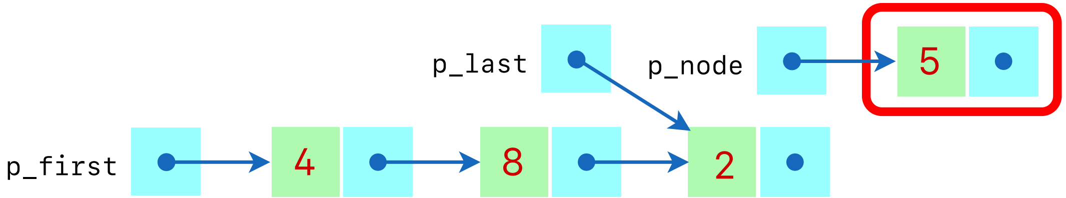

- Using a pointer (`p_last`), we start at the first node and traverse the structure until we reach the last node (its `p_next` pointer is `NULL`):



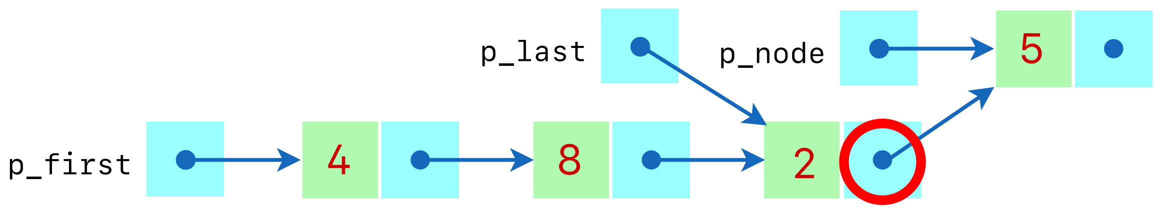

- A new node is allocated with element 5 and its `p_next` pointer is set to `NULL`:



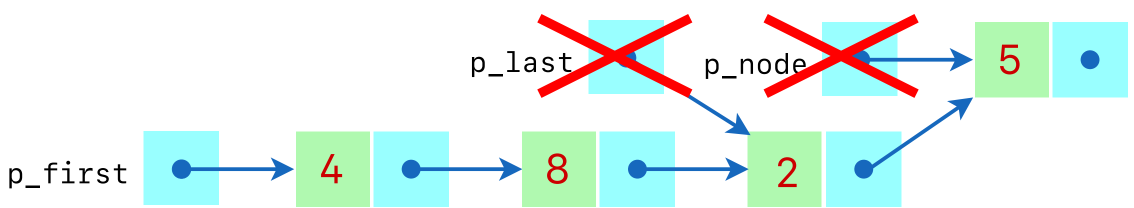
# Appending an Element at the End of a Non-Empty Linked Structure (II)




- The `p_next` pointer of the last node is updated to point to the new node:




- `p_last` and `p_node` are automatic variables so they are deallocated when the function returns:



## Inserting an Element at an Arbitrary Position in a Linked Structure

- Use pointer `p_current` to traverse the structure to the desired position.
- Use pointer `p_previous` to always track the node before `p_current`.

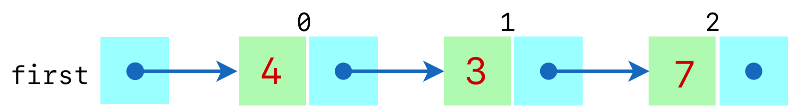
- Initially, `p_previous` is set to `NULL` and `p_current` points to the first node.
- On each iteration, update `p_previous` to the current node before moving `p_current` to the next node.

- The code to move `p_current` to the desired position (`index`) is as follows:

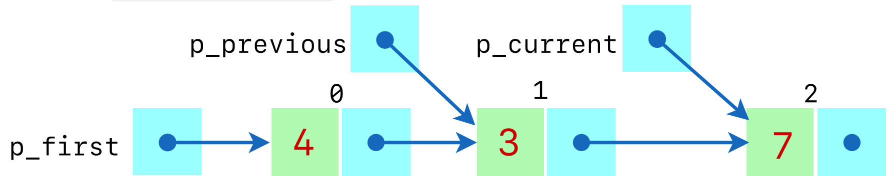
```
struct Node* p_previous = NULL;
struct Node* p_current = p_first;

for (size_t i = 0; i < index; i++) {
  assert(p_current != NULL && "Invalid index");
  p_previous = p_current;
  p_current = p_current->p_next;
}
```

# Inserting an Element at an Arbitrary Position in a Linked Structure (II)

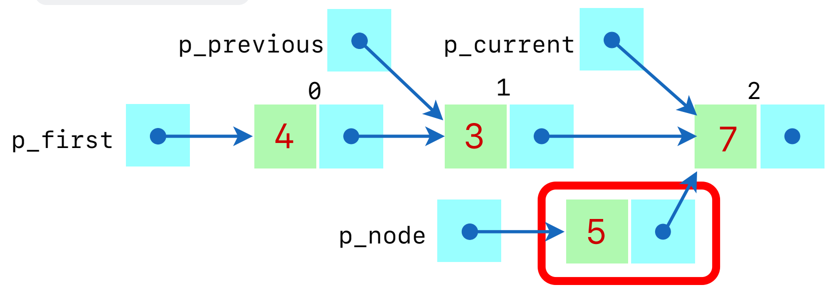
- Starting from this configuration, we are going to `insert` element 5 at position 2:



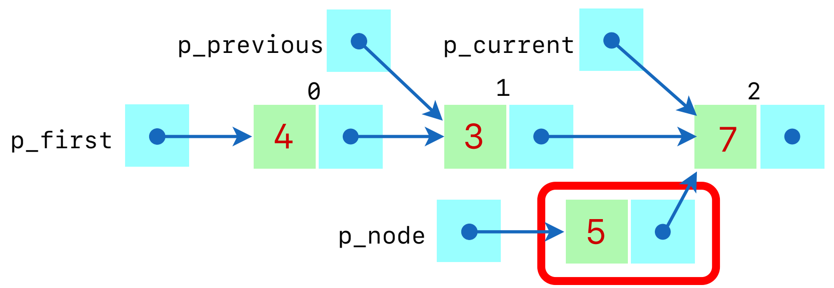

- We start at the first node and traverse the structure to move `p_current` to the node at position 2 and `p_previous` to the node before it:



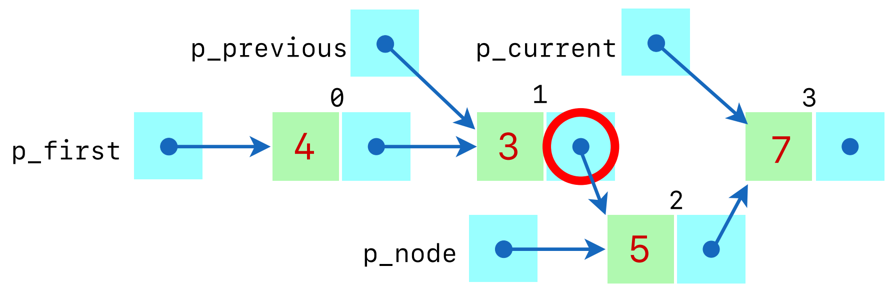

- A new node is allocated with the new element (5) and its `p_next` pointer is set to the node pointed by `p_current`:



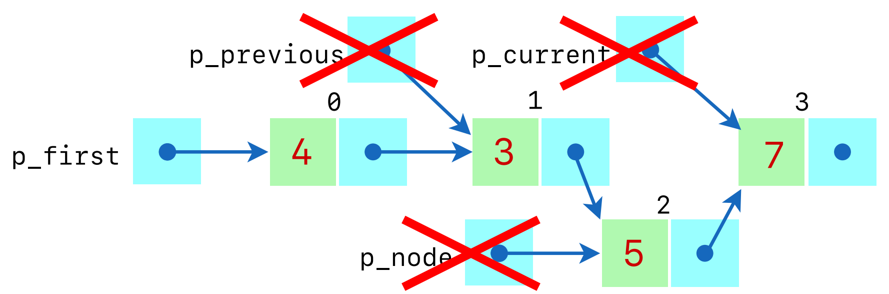
# Inserting an Element at an Arbitrary Position in a Linked Structure (III)




- The `p_next` pointer of the node pointed by `p_previous` is updated to point to the new node:



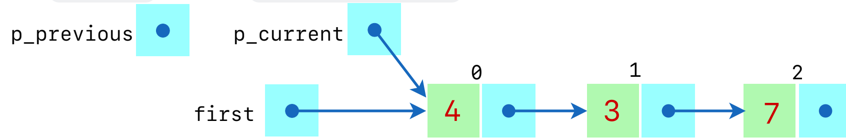

- `p_current`, `p_previous` and `p_node` are automatic variables so they are deallocated when the function returns:



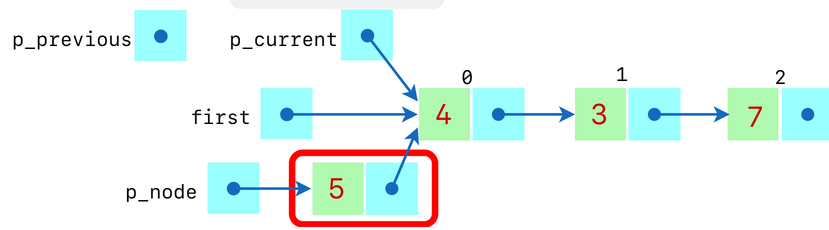
# Inserting an Element at an Arbitrary Position in a Linked Structure (IV)

## Special case: inserting at position 0

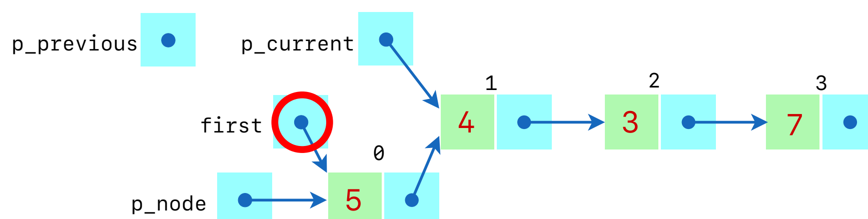
- If we want to insert an element at position 0, the loop will not be executed and `p_previous` will be `NULL` and `p_current` will point to the first node:




- As usual, we allocate a new node with the element to insert (5) and set its `p_next` pointer to the node pointed by `p_current`:



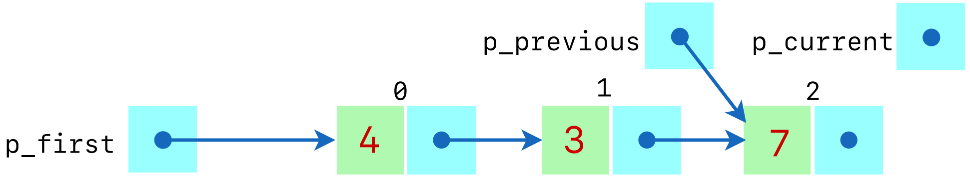

- In this case, we also need to update the pointer to the first node to point to the new node:



# Inserting an Element at an Arbitrary Position in a Linked Structure (V)

## Special case: inserting after the last node

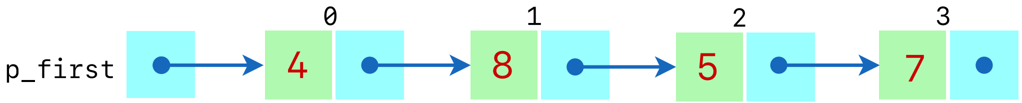
- If we want to insert an element at the end of the structure, the loop will traverse the entire structure, `p_previous` will point to the last node and `p_current` will be `NULL`.
- In this example we are going to insert element 5 at position 3:



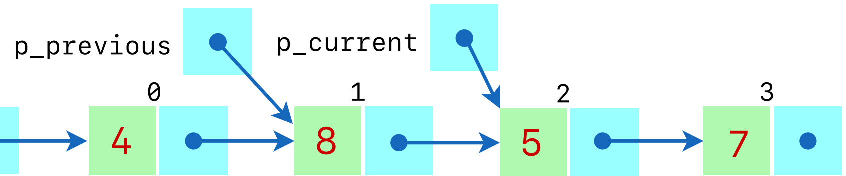

- In this case, the same algorithm that we used before in the general case can be used (you should verify that it works correctly).

# Deleting an Element at an Arbitrary Position in a Linked Structure

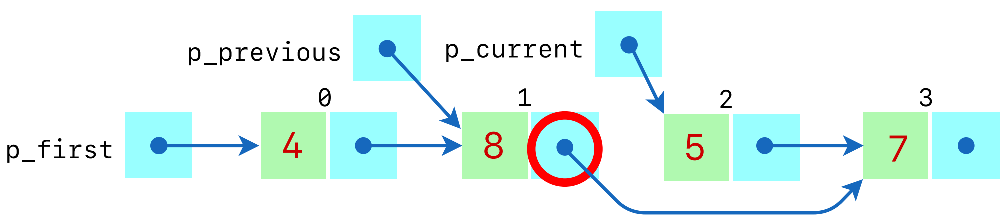
- Starting from this configuration, we are going to `delete` the element at position 2:



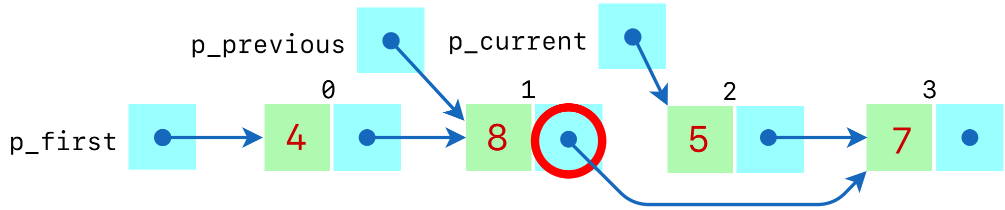

- We start at the first node and traverse the structure until we reach the node at position 2 with `p_current` and the previous node with `p_previous`:



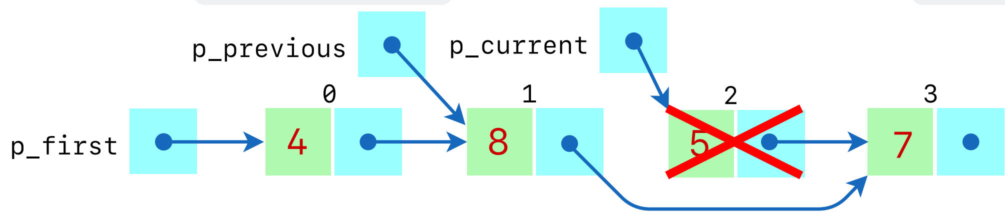

- The `p_next` pointer of the node pointed by `p_previous` is updated to point to the node pointed by the `p_next` component of `p_current`:



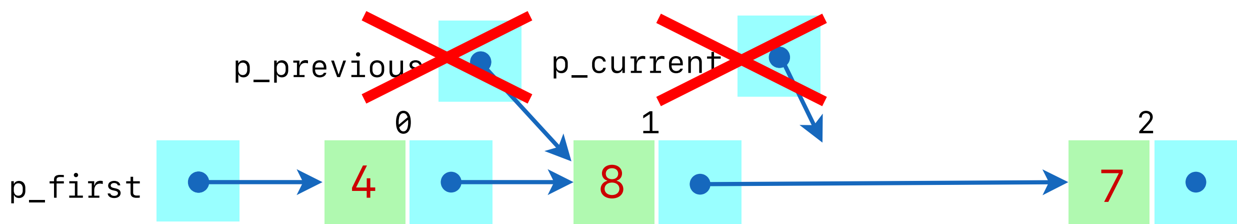
# Deleting an Element at an Arbitrary Position in a Linked Structure (II)




- The node pointed by `p_current` is deallocated using `free`:



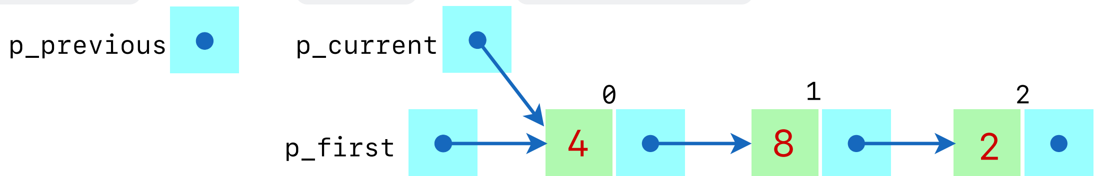

- `p_current` and `p_previous` are automatic variables so they are deallocated when the function returns:



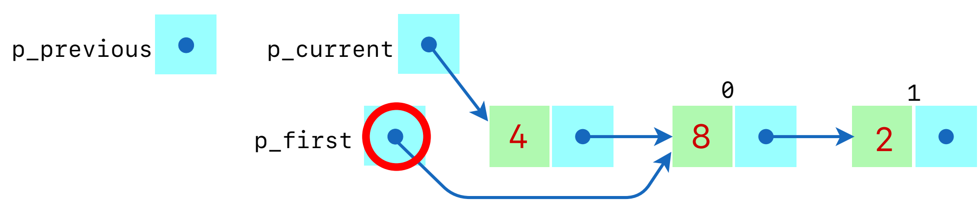
# Deleting an Element at an Arbitrary Position in a Linked Structure (III)

## Special case: deleting the first node

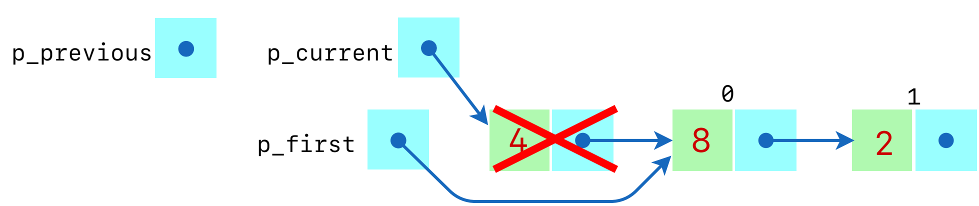
- If we want to delete the element at position 0, the loop will not be executed and `p_previous` will be `NULL` and `p_current` will point to the first node:




- In this case, we need to update the pointer to the first node to point to the node pointed by the `p_next` component of `p_current`:



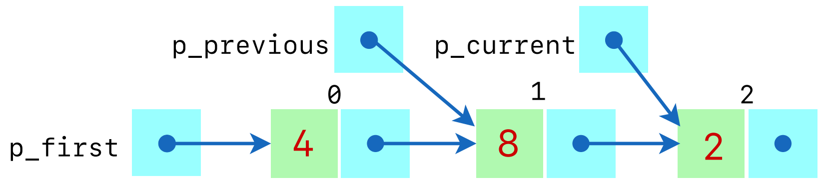

- And the node pointed by `p_current` must be deallocated using `free`:



# Deleting an Element at an Arbitrary Position in a Linked Structure (IV)

## Special case: deleting the last node

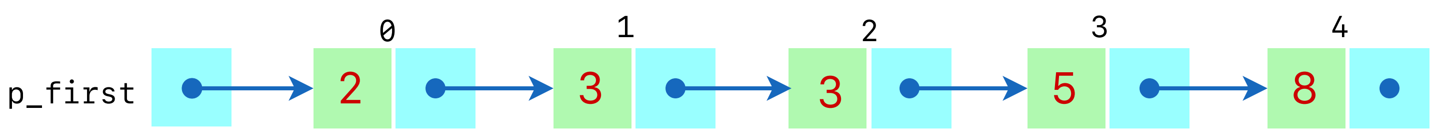
- If we want to delete the last node, the loop will traverse the entire structure, `p_previous` will point to the node before the last node and `p_current` will point to the last node.
- In this example we are going to delete the element at position 2:




- In this case, the same algorithm that we used before in the general case can be used (you should verify that it works correctly).

## Sorted Linked Structures

- A sorted linked structure is a linked structure where the elements are always stored in some specific order.
- Most of the operations are similar to those of a regular linked structure, except for the insertion.
- The insertion operation takes an element and inserts it in the correct position to maintain the sorted order.
- Here is an example of a sorted linked structure of integers in ascending order:



## Sorted Linked Structures. Exercise

Create a function named `sorted_insert` that accepts a pointer to a pointer to the first node of a sorted linked structure of integers and an integer element to insert. The function should insert the element in the correct position to maintain the structure's ascending order.

**Hint**: Utilize the `p_previous` and `p_current` pointers technique to find the appropriate insertion point for the new element.

## The `LinkedList` Data Structure

- Some disadvantages of representing a linked structure by a pointer to the first node are:
    - We need to pass a pointer to the pointer to the first node to every function that may modify the beginning of the structure. Working with pointers to pointers can be error-prone.
    - Computing the size of the structure requires traversing the entire structure (O(n) complexity).
    - Accessing the last node requires traversing the entire structure (O(n) complexity).

# The `LinkedList` Data Structure (II)

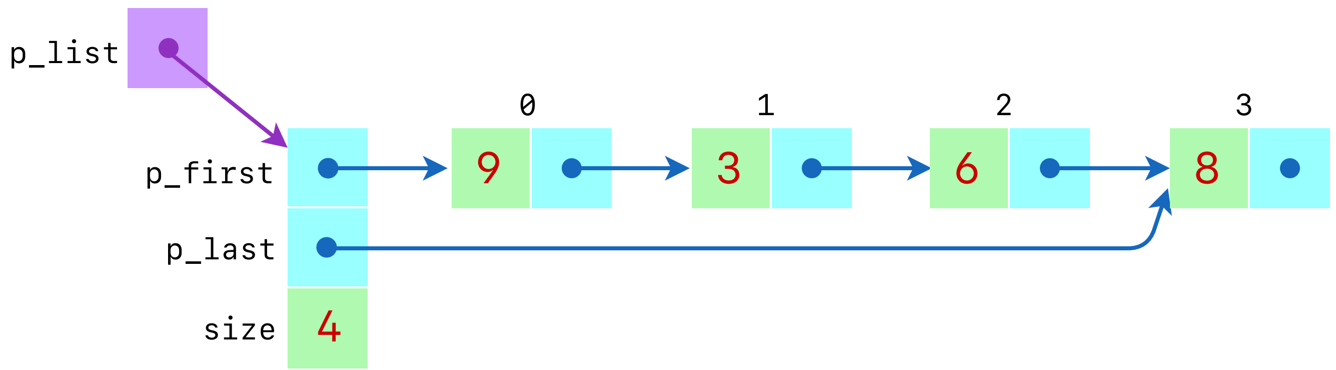
- To address these issues, we can define a `LinkedList` data structure that includes a pointer to the first node, a pointer to the last node, and a variable to track the size of the structure.

```c
// Node in a LinkedList
struct Node {
  int element;
  struct Node* p_next;
};

// LinkedList implementation
struct LinkedList {
  struct Node* p_first;
  struct Node* p_last;
  size_t size;
};
```

- We will pass a pointer to a `LinkedList` structure to functions that operate on the linked structure. This way, we can access and manipulate the first node, last node, and size of the structure directly.

# The `LinkedList` Data Structure (III)

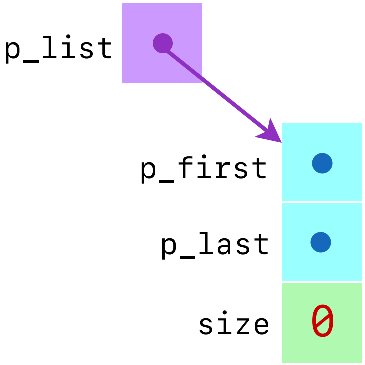
- An example of a `LinkedList` containing 4 integers is shown below:




- **Invariants**:
  - The `p_first` pointer points to the first node.
  - The `p_last` pointer points to the last node.
  - Each node's `p_next` pointer points to the next node in the structure.
  - The last node's `p_next` pointer is `NULL`.
  - The `size` variable is the number of elements in the structure.

# The `LinkedList` Data Structure (IV)

- An example of an empty `LinkedList` is shown below:



# The `LinkedList.h` Header File

- All functions now receive a pointer to a `LinkedList`. The only exception is `LinkedList_free`, which receives a pointer to a pointer to a `LinkedList` to allow the function to set the pointer to `NULL`, after freeing the structure, in order to avoid dangling pointers.

```c
#ifndef LinkedList_H // conditional compilation directive
#define LinkedList_H // avoids multiple inclusion of the header file

#include <stdbool.h>
#include <stddef.h>

struct Node { // Node in a LinkedList
  int element;
  struct Node* p_next;
};

struct LinkedList { // LinkedList implementation
  struct Node* p_first;
  struct Node* p_last;
  size_t size;
};

struct LinkedList* LinkedList_new();
void LinkedList_free(struct LinkedList** p_p_list);
struct LinkedList* LinkedList_copyOf(const struct LinkedList* p_list);
bool LinkedList_isEmpty(const struct LinkedList* p_list);
size_t LinkedList_size(const struct LinkedList* p_list);
void LinkedList_clear(struct LinkedList* p_list);
void LinkedList_append(struct LinkedList* p_list, int element);
void LinkedList_prepend(struct LinkedList* p_list, int element);
void LinkedList_insert(struct LinkedList* p_list, size_t index, int element);
int LinkedList_get(const struct LinkedList* p_list, size_t index);
void LinkedList_set(const struct LinkedList* p_list, size_t index, int element);
void LinkedList_delete(struct LinkedList* p_list, size_t index);
void LinkedList_print(const struct LinkedList* p_list);
#endif
```

# The `LinkedList.c` File

```c
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#include "LinkedList.h"

// Constructor for a new Node
static struct Node* Node_new(int element, struct Node* p_next) {
  struct Node* p_node = malloc(sizeof(struct Node));
  assert(p_node != NULL && "Node_new: not enough memory");

  p_node->element = element;
  p_node->p_next = p_next;
  return p_node;
}

// Destructor for a Node
static void Node_free(struct Node** p_p_node) {
  free(*p_p_node);
  *p_p_node = NULL;
}

// Constructor for a new empty LinkedList
struct LinkedList* LinkedList_new() {
  struct LinkedList* p_list = malloc(sizeof(struct LinkedList));
  assert(p_list != NULL && "LinkedList_new: not enough memory");

  p_list->p_first = NULL;
  p_list->p_last = NULL;
  p_list->size = 0;
  return p_list;
}
... // rest of the implementation
```

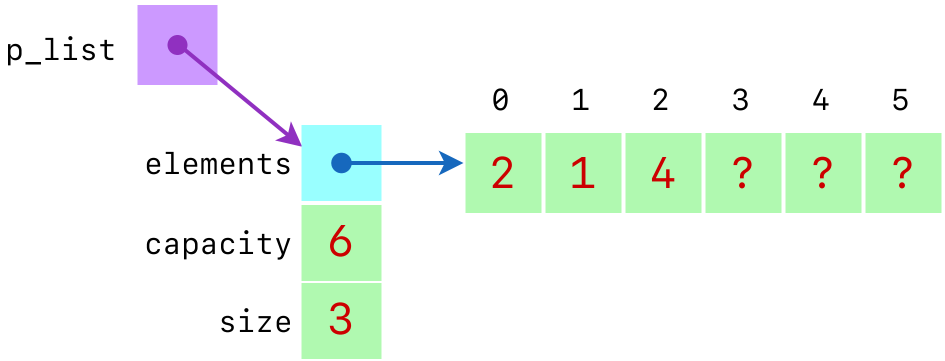
# The `LinkedList` Data Structure. Exercise

- Complete the implementation of the `LinkedList` data structure by implementing the functions declared in the `LinkedList.h` header file and any other private functions needed.
- Each operation should be implemented in a way that maintains the integrity (invariants) of the linked structure: `p_first` should point to the first node in the linked structure, `p_last` should point to the last node, and `size` should be the number of elements in the structure.
- Test the implementation by writing a program that uses the `LinkedList` data structure to store a list of integers and performs various operations on it.

# ArrayList

# The `ArrayList` Data Structure

- An `ArrayList` is a dynamic array that can grow to accommodate new elements as needed.
- Initially, it has the ability to hold a certain number of elements, known as its capacity. However, this capacity is not fixed and can be expanded dynamically as required.
- Each element's position in the `ArrayList` corresponds directly to its index in the array, with the element at list index `i` being placed at array index `i`.
- Inserting new elements into the `ArrayList` prompts a rightward **shift** of subsequent array elements, creating the necessary space.
- Conversely, when an element is removed, the subsequent elements **shift** leftward, eliminating any gaps.
- The implementation also maintains an integer variable `size` to keep track of the number of elements in the `ArrayList`.
- We are going to define a module for an `ArrayList` of integers. Its implementation looks like this:



# The `ArrayList.h` Header File

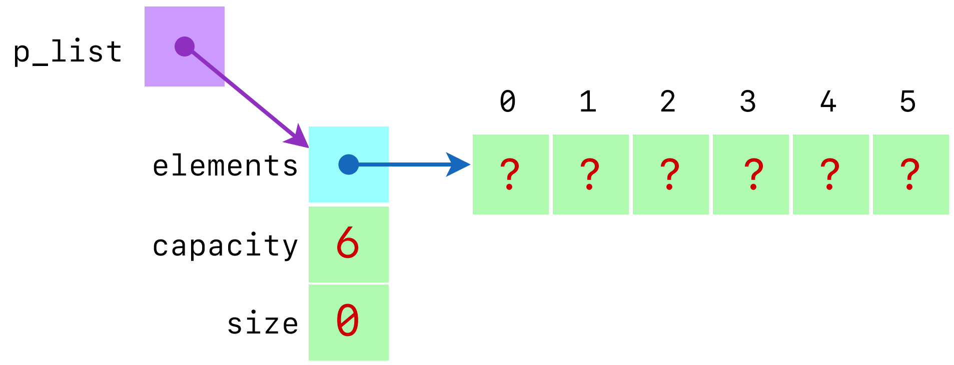
```c
#ifndef ARRAYLIST_H // conditional compilation directive
#define ARRAYLIST_H // avoids multiple inclusion of the header file

#include <stdbool.h>
#include <stddef.h>

// Structure corresponding to an ArrayList
struct ArrayList {
  int* elements;     // heap allocated array of elements
  size_t capacity;   // capacity of array
  size_t size;       // number of elements in array
};

struct ArrayList* ArrayList_new(size_t initialCapacity);
void ArrayList_free(struct ArrayList** p_p_list);
struct ArrayList* ArrayList_copyOf(const struct ArrayList* p_list);
bool ArrayList_isEmpty(const struct ArrayList* p_list);
size_t ArrayList_size(const struct ArrayList* p_list);
void ArrayList_clear(struct ArrayList* p_list);
void ArrayList_append(struct ArrayList* p_list, int element);
void ArrayList_prepend(struct ArrayList* p_list, int element);
void ArrayList_insert(struct ArrayList* p_list, size_t index, int element);
int ArrayList_get(const struct ArrayList* p_list, size_t index);
void ArrayList_set(const struct ArrayList* p_list, size_t index, int element);
void ArrayList_delete(struct ArrayList* p_list, size_t index);
void ArrayList_print(const struct ArrayList* p_list);
#endif
```

# The `ArrayList.c` File

```c
#include <assert.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#include "ArrayList.h"

// Constructor for a new empty ArrayList with the given initial capacity.
struct ArrayList* ArrayList_new(size_t initialCapacity) {
  assert(initialCapacity > 0 && "ArrayList_new: initialCapacity must be greater than 0");

  struct ArrayList* p_list = malloc(sizeof(struct ArrayList));
  assert(p_list != NULL && "ArrayList_new: not enough memory");

  p_list->elements = malloc(sizeof(int) * initialCapacity);
  assert(p_list->elements != NULL && "ArrayList_new: not enough memory");

  p_list->size = 0;
  p_list->capacity = initialCapacity;
  return p_list;
}
... // rest of the implementation
```

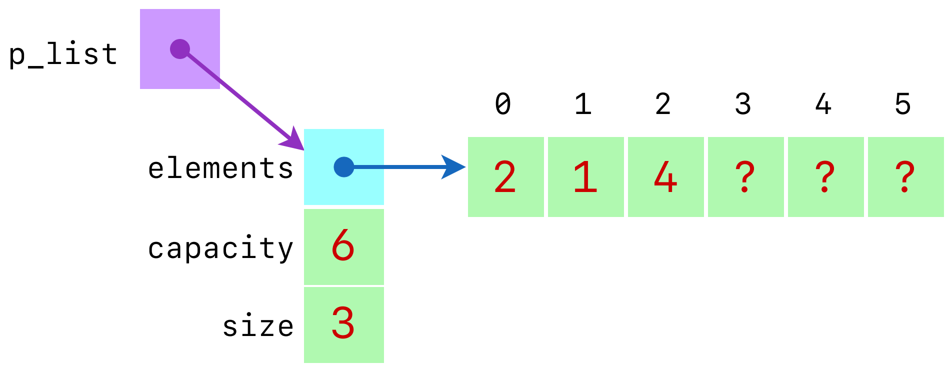
## `ArrayList` After Initialization

- Assuming initial capacity is 6, the array of elements has been allocated on the heap. When an array is allocated with `malloc`, its elements will contain garbage values (represented by ?):

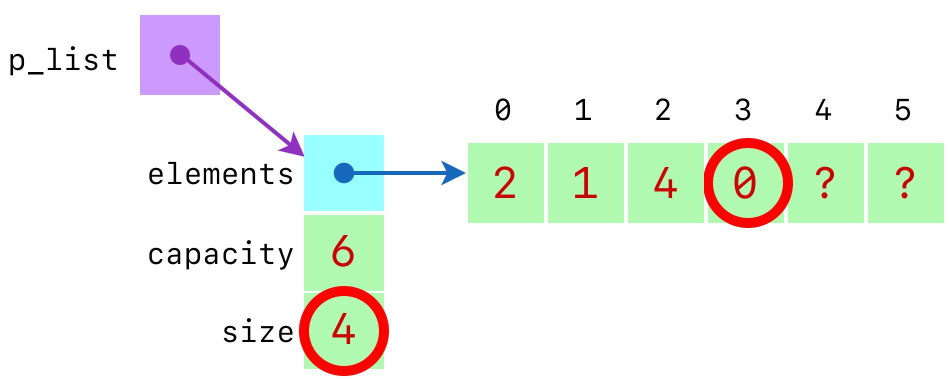


# Appending an Element at the End in `ArrayList`

- `ArrayList_append` adds an element after the last element in the list:
  - Capacity of the array to accommodate the new element must be ensured.
  - The array stores the new element at the index designated by `size`.
  - `size` is incremented to keep track of the number of elements in the list.
- Starting from this configuration, we are going to append element `0` to the `ArrayList`:




- After appending element `0`:



## `realloc` Function in C

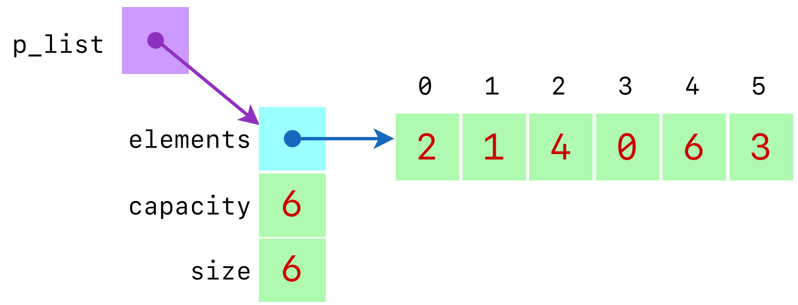
- **Purpose**: The `realloc` function is used to resize a memory block that was previously allocated with `malloc` or `calloc`.
- **Declaration**:

```
void* realloc(void* ptr, size_t new_size);
```

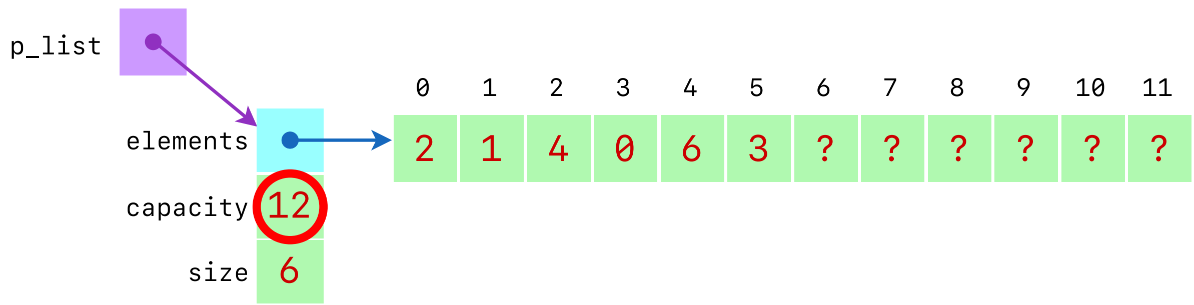
- **Functionality**:
  - Changes the size of the memory block pointed to by `ptr` to `new_size` bytes.
  - Preserves the content of the original memory block up to the minimum of the old and new sizes.
  - If `new_size` is larger than the old size, the additional memory is uninitialized.
- **Usage Notes**:
  - You do not need to free the original memory block before calling `realloc`.
  - The function returns a pointer to the reallocated memory block.
  - If reallocation fails, it returns `NULL` and the original memory block remains unchanged.

## Ensuring Capacity in `ArrayList`

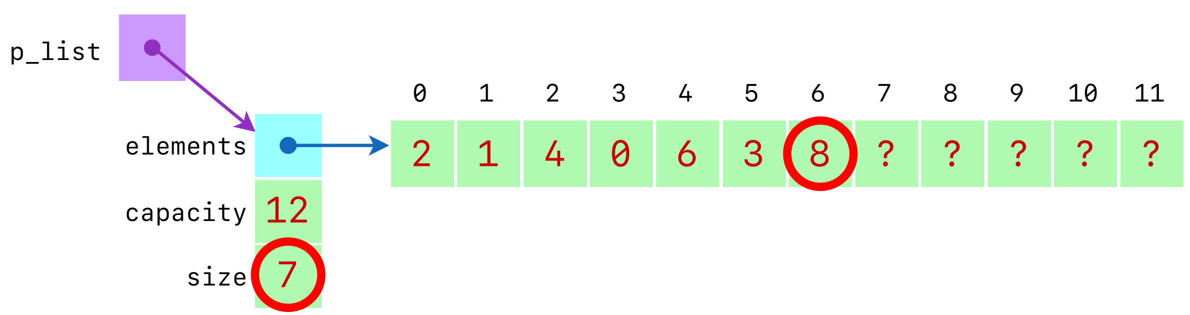
- Starting from this configuration, we are going to append element 8 to the `ArrayList`:




- We reallocate `elements` using `realloc` to double its capacity:

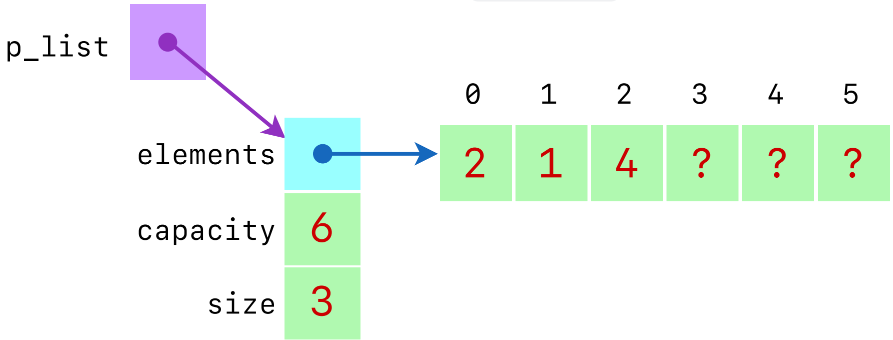



- Now we can append element 8:

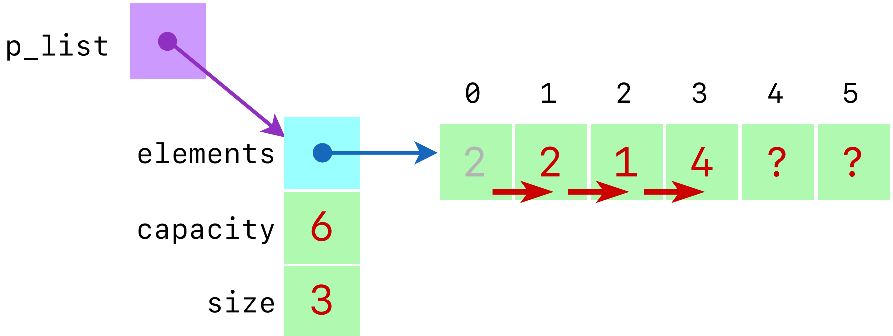


## Prepending an Element at the Beginning in `ArrayList`

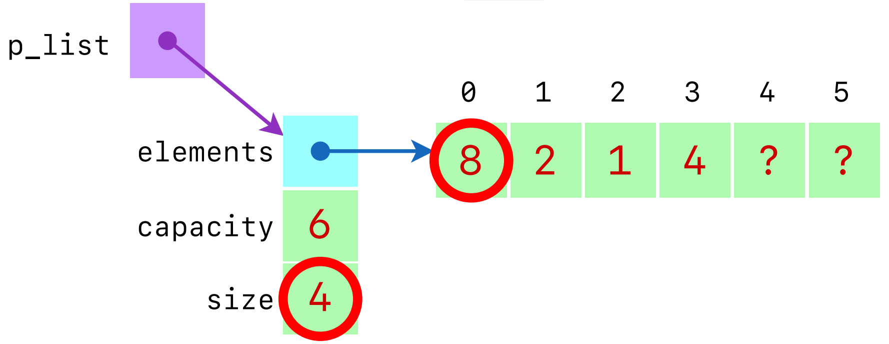
- Starting from this configuration, we are going to `prepend` element 8 at the beginning:




- The elements at positions 2, 1 and 0 are shifted rightward to make space for the new element:

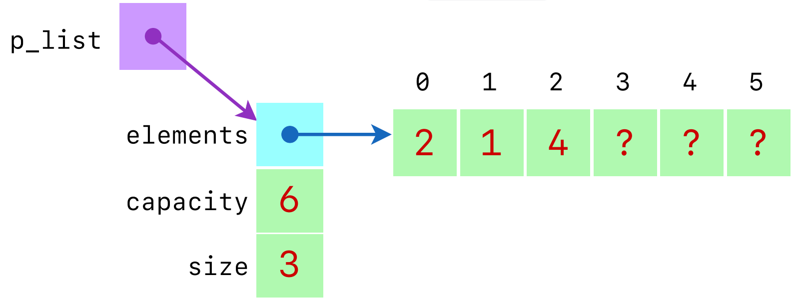



- The new element (8) is stored at position 0 and `size` is incremented:

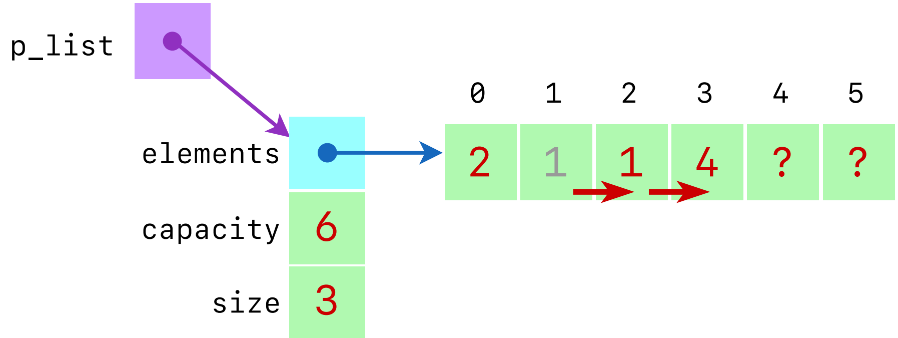


# Inserting an Element at an Arbitrary Position in `ArrayList`

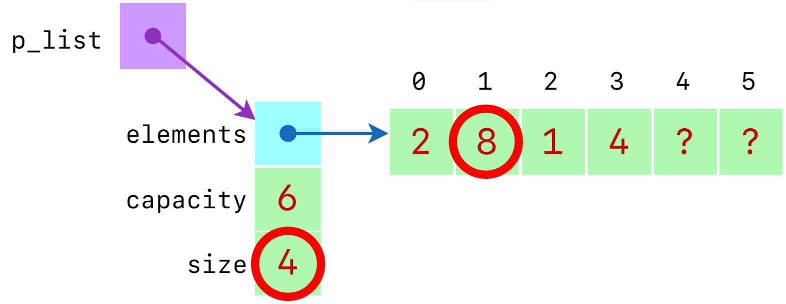
- Starting from this configuration, we are going to `insert` the element 8 at position 1:




- The elements at positions 2 and 1 are shifted rightward to make space for the new element:

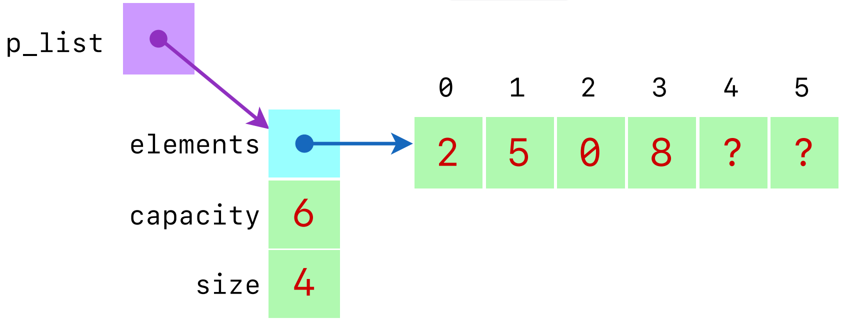



- The new element (8) is stored at position 1 and `size` is incremented:

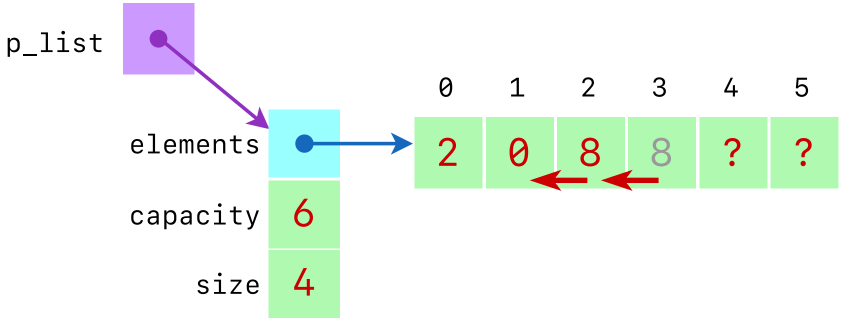


# Deleting an Element at an Arbitrary Position in `ArrayList`

- Starting from this configuration, we are going to `delete` the element at position 1:




- The elements at positions 2 and 3 are shifted leftward to fill the gap left by the removed element:




- `size` is decremented :