Laboratorio de Programación

2016 -2

Juan Camilo Rada[1]

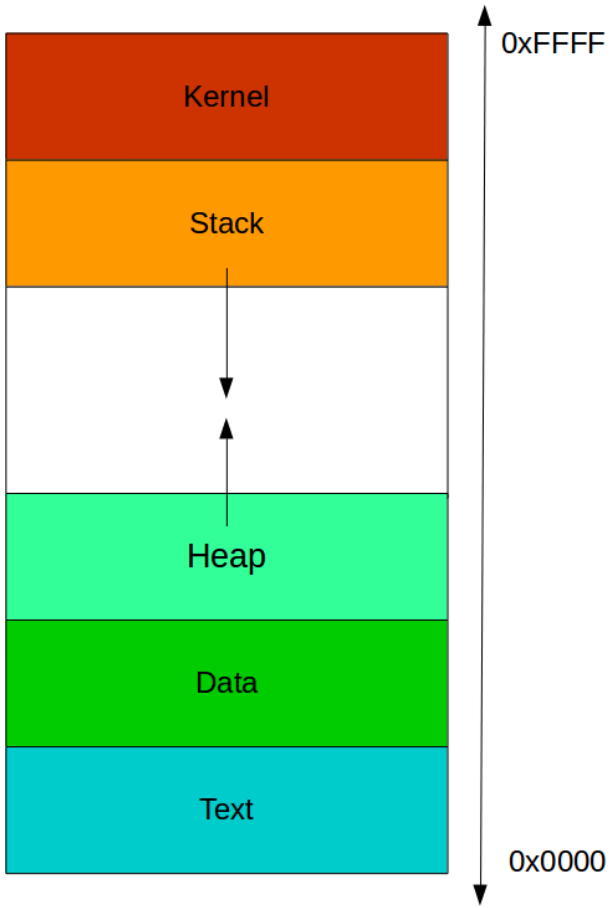
[1]Departamento de Electronica y ciencias de la computación

Septiembre de 2016

Outline

Outline

Program memory



Dynamic Memory Allocation

Static

- The compiler determines the memory that needs to be allocated

Dynamic Memory Allocation

Static

- The compiler determines the memory that needs to be allocated
- The memory allocation is made in execution time

Dynamic Memory Allocation

Static

- The compiler determines the memory that needs to be allocated
- The memory allocation is made in execution time
- `double[5] b;`
  We need to specify the size of the array to be allocated

Dynamic Memory Allocation

What if ...

... We are reading a file that may contain hundreds, thousands or just one character

- `char[5000] chars;`
  We are expecting the file to contain less than 5000 chars!

Dynamic Memory Allocation

What if ...

... We are reading a file that may contain hundreds, thousands or just one character

- `char[5000] chars;`
  We are expecting the file to contain less than 5000 chars!
- What if the file contains 5001 chars?

Dynamic Memory Allocation

What if ...

... We are reading a file that may contain hundreds, thousands or just one character

- `char[5000] chars;`
  We are expecting the file to contain less than 5000 chars!
- What if the file contains 5001 chars?
- We need to *allocate* more memory

Dynamic Memory Allocation

What if ...

... We are reading a file that may contain hundreds, thousands or just one character

- `char[5000] chars;`
  We are expecting the file to contain less than 5000 chars!
- What if the file contains 5001 chars?
- We need to *allocate* more memory
- What if the file contains just only 1 char?

Dynamic Memory Allocation

What if ...

... We are reading a file that may contain hundreds, thousands or just one character

- `char[5000] chars;`
  We are expecting the file to contain less than 5000 chars!
- What if the file contains 5001 chars?
- We need to *allocate* more memory
- What if the file contains just only 1 char?
- We need to *free* memory

Dynamic Memory Allocation

Dynamic Memory

- The ability to allocate or release memory in execution time

Dynamic Memory Allocation

Dynamic Memory

- The ability to allocate or release memory in execution time
- We can allocate new memory if we need to store more data in memory

Dynamic Memory Allocation

Dynamic Memory

- The ability to allocate or release memory in execution time
- We can allocate new memory if we need to store more data in memory
- We can release the memory that is not needed anymore!

Dynamic Memory Allocation

Dynamic Memory

- The ability to allocate or release memory in execution time
- We can allocate new memory if we need to store more data in memory
- We can release the memory that is not needed anymore!
- Allocated memory is accessed using Pointers

Dynamic Memory Allocation

Dynamic Memory

- The ability to allocate or release memory in execution time
- We can allocate new memory if we need to store more data in memory
- We can release the memory that is not needed anymore!
- Allocated memory is accessed using Pointers
- `stdlib.h` provide functions to work with dynamic memory

Outline

Dynamic Memory Allocation

```
void* malloc (size_t size);
```

The `malloc` function

- Allocate a new block of memory

Dynamic Memory Allocation

```
void* malloc (size_t size);
```

The `malloc` function

- Allocate a new block of memory
- `size_t size`
  The size of the memory block to be allocated

Dynamic Memory Allocation

```
void* malloc (size_t size);
```

The `malloc` function

- Allocate a new block of memory
- `size_t size`
  The size of the memory block to be allocated
- `void*`
  Returns a pointer to the allocated block of memory

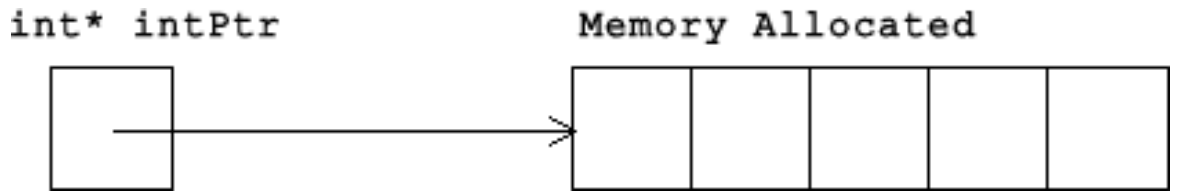
Dynamic Memory Allocation

```
void* malloc (size_t size);
```

The `malloc` function

- Allocate a new block of memory
- `size_t size`
  The size of the memory block to be allocated
- `void*`
  Returns a pointer to the allocated block of memory
- Returns a `NULL` pointer if the function failed to allocate memory

Dynamic Memory Allocation



The `malloc` function

- Allocate a new block of memory
- `int size`
  The size of the memory block to be allocated
- `void*`
  Returns a pointer to the allocated block of memory
- Returns a `NULL` pointer if the function failed to allocate memory

Dynamic Memory Allocation

Test the code

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

int main() {
    int pos;
    int i = 0;
    printf("Positions to be allocated ");
    scanf("%d", &pos);

    int* intPtr = (int* ) malloc(sizeof(int)*pos);

    for(i=0; i<pos; i++)
        intPtr[i] = 20;

    for(i=0; i<pos; i++)
        printf("int[%i] = %i\n", i, intPtr[i]);
}
```

Dynamic Memory Allocation

```
int* intPtr = (int* ) malloc(sizeof(int)*pos);
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer

Dynamic Memory Allocation

```
int* intPtr = (int* ) malloc(sizeof(int)*pos);
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `sizeof(int)*pos`
  The size of the new block of memory $\rightarrow$ The size of `pos` integers

Dynamic Memory Allocation

```
int* intPtr = (int* ) malloc(sizeof(int)*pos);
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `sizeof(int)*pos`
  The size of the new block of memory $\rightarrow$ The size of `pos` integers
- The pointer `initPtr` points to the new allocated memory

Dynamic Memory Allocation

```
int* intPtr = (int* ) malloc(sizeof(int)*pos);
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `sizeof(int)*pos`
  The size of the new block of memory → The size of `pos` integers
- The pointer `initPtr` points to the new allocated memory
- We use the `initPtr` to move along the block of memory

Dynamic Memory Allocation

```
void* calloc (int num, size_t size);
```

The `calloc` function

- Allocate new array with `num` positions, each of them `size` bytes long

Dynamic Memory Allocation

```
void* calloc (int num, size_t size);
```

The `calloc` function

- Allocate new array with `num` positions, each of them `size` bytes long
- `int num`
  The number of positions which the array will contain

Dynamic Memory Allocation

```
void* calloc (int num, size_t size);
```

The `calloc` function

- Allocate new array with `num` positions, each of them `size` bytes long
- `int num`
  The number of positions which the array will contain
- `size_t size`
  The size in bytes per block

Dynamic Memory Allocation

```
void* calloc (int num, size_t size);
```

The `calloc` function

- Allocate new array with `num` positions, each of them `size` bytes long
- `int num`
  The number of positions which the array will contain
- `size_t size`
  The size in bytes per block
- `void*`
  Returns a pointer to the allocated block of memory

Dynamic Memory Allocation

```
void* calloc (int num, size_t size);
```

The `calloc` function

- Allocate new array with `num` positions, each of them `size` bytes long
- `int num`
  The number of positions which the array will contain
- `size_t size`
  The size in bytes per block
- `void*`
  Returns a pointer to the allocated block of memory
- Returns a `NULL` pointer if the function failed to allocate memory

Dynamic Memory Allocation

Test the code

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

int main() {
    int pos;
    int i = 0;
    printf("Positions to be allocated ");
    scanf("%d", &pos);

    int* intPtr = (int* ) calloc(pos, sizeof(int));

    for(i=0; i<pos; i++)
        intPtr[i] = 20;

    for(i=0; i<pos; i++)
        printf("int[%i] = %i\n", i, intPtr[i]);
}
```

Dynamic Memory Allocation

```
int* intPtr = (int* ) calloc(pos, sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer

Dynamic Memory Allocation

```
int* intPtr = (int* ) calloc(pos, sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `sizeof(int)`
  The size per position $\rightarrow$ The size of `int`

Dynamic Memory Allocation

```
int* intPtr = (int* ) calloc(pos, sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `sizeof(int)`
  The size per position $\rightarrow$ The size of `int`
- The pointer `initPtr` points to the new allocated array

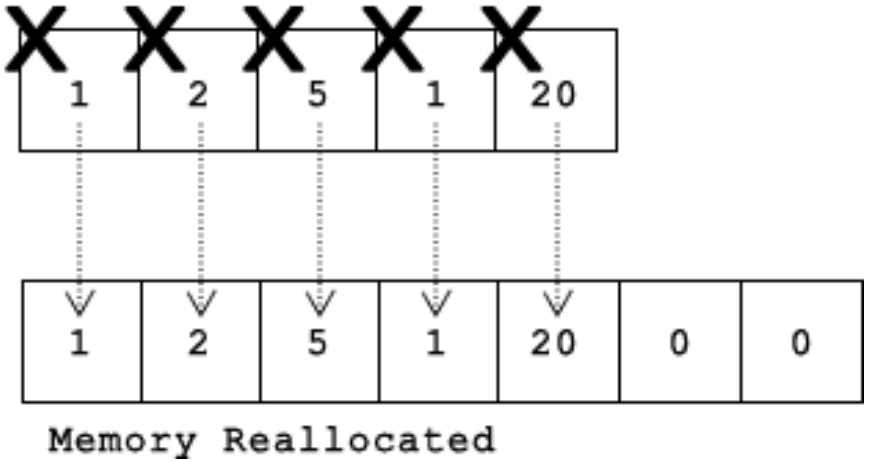
Dynamic Memory Allocation

```
int* intPtr = (int* ) calloc(pos, sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `sizeof(int)`
  The size per position $\rightarrow$ The size of `int`
- The pointer `initPtr` points to the new allocated array
- We use the `initPtr` to move along the array

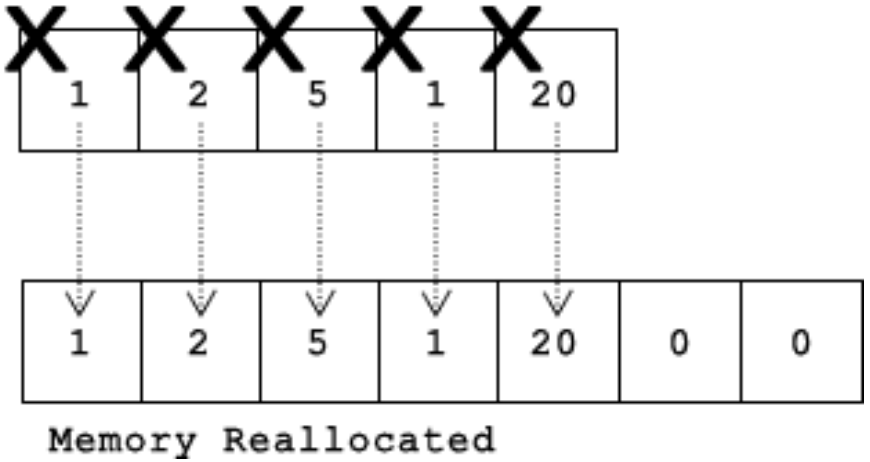
Dynamic Memory Allocation



The `realloc` function

- Helps to allocate more memory if needed

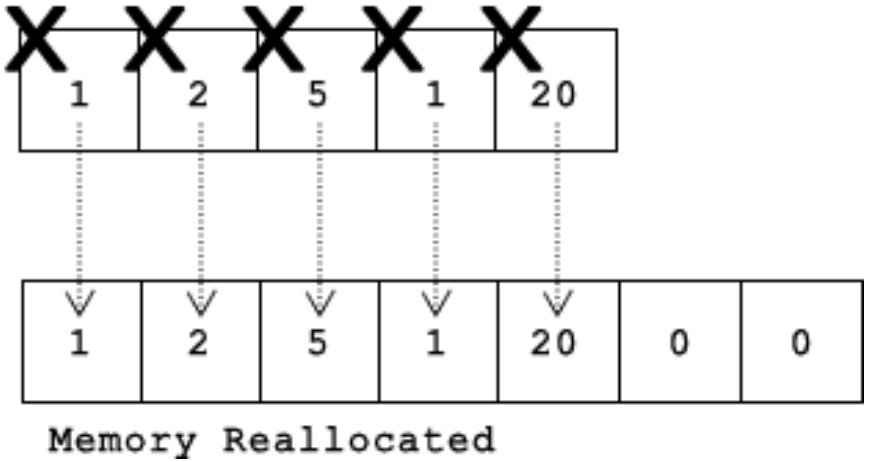
Dynamic Memory Allocation



The `realloc` function

- Helps to allocate more memory if needed
- Looks for block of memory which fits all memory required

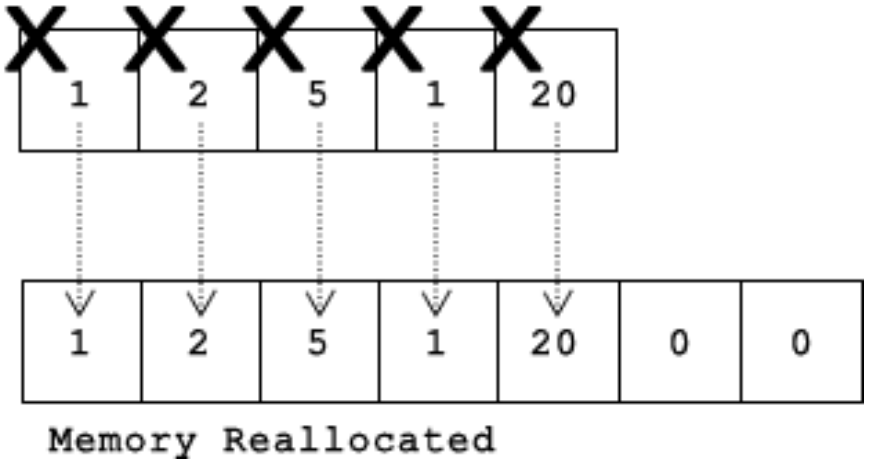
Dynamic Memory Allocation



The `realloc` function

- Helps to allocate more memory if needed
- Looks for block of memory which fits all memory required
- Copy all the data from the old memory positions, to the new one

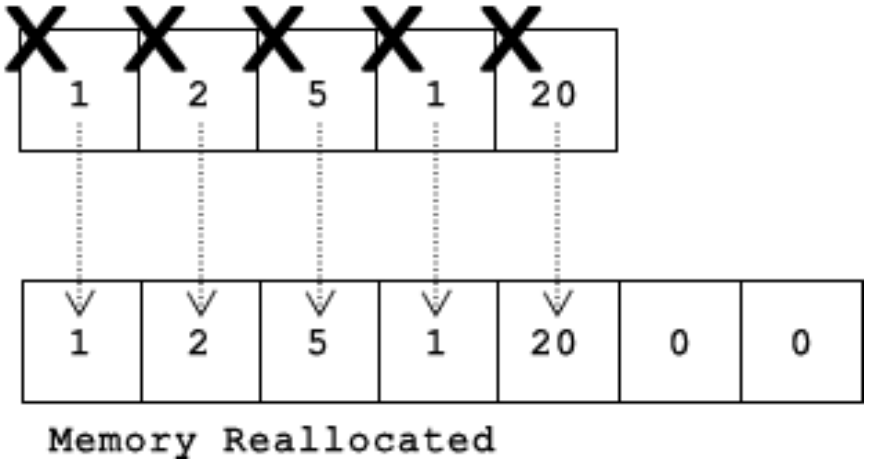
Dynamic Memory Allocation



The `realloc` function

- Helps to allocate more memory if needed
- Looks for block of memory which fits all memory required
- Copy all the data from the old memory positions, to the new one
- Move the pointer to point the new allocated memory

Dynamic Memory Allocation



The `realloc` function

- Helps to allocate more memory if needed
- Looks for block of memory which fits all memory required
- Copy all the data from the old memory positions, to the new one
- Move the pointer to point the new allocated memory

Dynamic Memory Allocation

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

The realloc function

- Reallocate a block of memory

Dynamic Memory Allocation

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

The `realloc` function

- Reallocate a block of memory
- `void* ptr`
  Pointer to the memory block previously allocated, that will be moved

Dynamic Memory Allocation

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

The `realloc` function

- Reallocate a block of memory
- `void* ptr`
  Pointer to the memory block previously allocated, that will be moved
- `size_t size`
  The size in bytes for the new memory block

Dynamic Memory Allocation

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

The `realloc` function

- Reallocate a block of memory
- `void* ptr`
  Pointer to the memory block previously allocated, that will be moved
- `size_t size`
  The size in bytes for the new memory block
- `void*`
  Returns a pointer to the reallocated block of memory

Dynamic Memory Allocation

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

The `realloc` function

- Reallocate a block of memory
- `void* ptr`
  Pointer to the memory block previously allocated, that will be moved
- `size_t size`
  The size in bytes for the new memory block
- `void*`
  Returns a pointer to the reallocated block of memory
- Returns a `NULL` pointer if the function failed to allocate memory

Dynamic Memory Allocation

Test the code

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int* ptr = NULL;
    int count = 0, number = 0, ver = 1, i = 0;
    while(ver)
    {
        printf("Give me an integer: ");
        scanf("%d", &number);
        count++;
        ptr = (int*) realloc(ptr, count*sizeof(int));
        ptr[count-1] = number;

        printf("Do you want more numbers 1/0? ");
        scanf("%d", &ver);
    }
    for(i=0; i<count; i++)
        printf("int[%i] = %i\n", i, ptr[i]);
```

Dynamic Memory Allocation

```
ptr = (int*) realloc(ptr, count*sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer

Dynamic Memory Allocation

```
ptr = (int*) realloc(ptr, count*sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `count*sizeof(int)`
  The size of the new block of memory $\rightarrow$ The size of `count` integers

Dynamic Memory Allocation

```
ptr = (int*) realloc(ptr, count*sizeof(int));
```

Statement

- `(int* )`
  Casts the `void*` pointer to an integer pointer
- `count*sizeof(int)`
  The size of the new block of memory $\rightarrow$ The size of `count` integers
- The pointer `ptr` points to the old allocated memory, then the new one

Dynamic Memory Allocation

```
void free (void* ptr);
```

The free function

- Deallocate a block of memory, making it available for future use

Dynamic Memory Allocation

```
void free (void* ptr);
```

The `free` function

- Deallocate a block of memory, making it available for future use
- `void* ptr`
  Pointer to the block of memory to be deallocated
- The block of memory to be deallocated, had to be allocated with `malloc`, `calloc` or `realloc`, otherwise it will face an unexpected behaviour

Outline

Dynamic Memory Allocation - Matrix

Matrix

- We can also allocate matrices with dynamic memory

Dynamic Memory Allocation - Matrix

Matrix

- We can also allocate matrices with dynamic memory
- `int** ptr`
  Double pointers helps to accomplish the allocation!

Dynamic Memory Allocation - Matrix

Matrix

- We can also allocate matrices with dynamic memory
- `int** ptr`
  Double pointers helps to accomplish the allocation!
- *A double pointer is a pointer which points to an array of pointers!*

Dynamic Memory Allocation - Matrix

Matrix

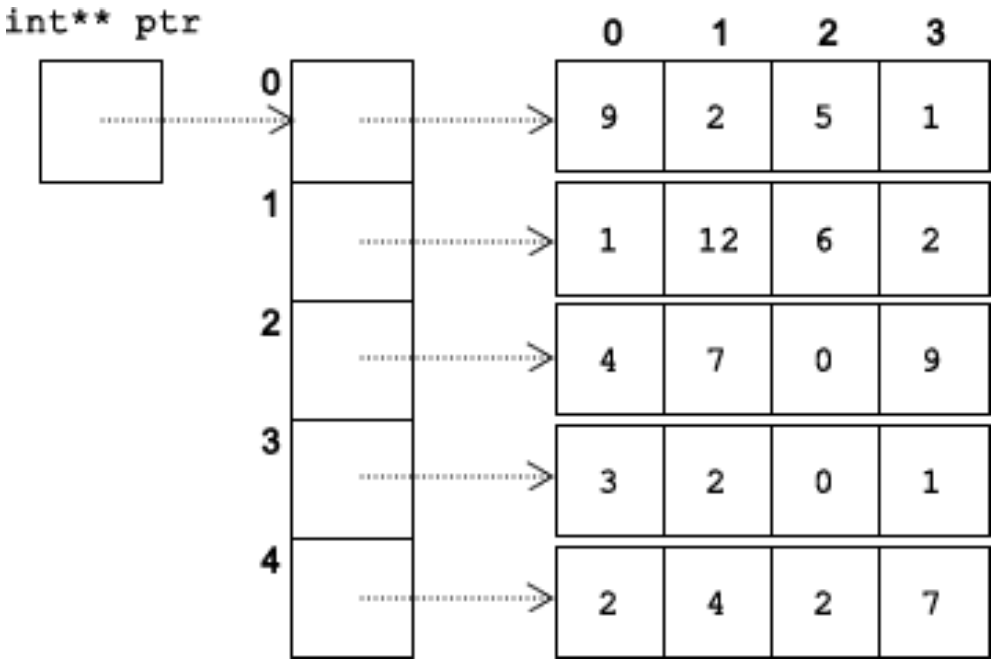
- We can also allocate matrices with dynamic memory
- `int** ptr`
  Double pointers helps to accomplish the allocation!
- *A double pointer is a pointer which points to an array of pointers!*
- The pointers in the *Array of pointers* are allocated using `malloc`, `calloc` or `realloc`

Dynamic Memory Allocation - Matrix

Matrix

- We can also allocate matrices with dynamic memory
- `int** ptr`
  Double pointers helps to accomplish the allocation!
- *A double pointer is a pointer which points to an array of pointers!*
- The pointers in the *Array of pointers* are allocated using `malloc`, `calloc` or `realloc`
- `ptr[0][2]`
  We can access every position with the double bracket operator

Dynamic Memory Allocation - Matrix



Dynamic Memory Allocation - Matrix

```
int** ptr = calloc(rows, sizeof(int));
for(i=0; i<rows; i++)
    ptr[i] = calloc(cols, sizeof(int));
```

Statement

1. `int** ptr = calloc(rows, sizeof(int));`
   The array of pointers has to be allocated
2. `ptr[i] = calloc(cols, sizeof(int));`
   We allocate every row as a dynamic array

Dynamic Memory Allocation

Test the code

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

int main() {
    int i=0, rows = 5, cols = 4;

    int** ptr = calloc(rows, sizeof(int));
    for(i=0; i<rows; i++)
        ptr[i] = calloc(cols, sizeof(int));

    ptr[0][4] = 35;

    printf("%i", ptr[0][4]);
}
```

Dynamic Memory Allocation

Let's write some code

Using the `malloc`, `calloc` and `realloc` functions, write the next procedures

- `addColumn` - Add a new column to the dynamic matrix

Dynamic Memory Allocation

Let's write some code

Using the `malloc`, `calloc` and `realloc` functions, write the next procedures

- `addColumn` - Add a new column to the dynamic matrix
- `addRow` - Add a new row to the dynamic matrix

Dynamic Memory Allocation

Let's write some code

Using the `malloc`, `calloc` and `realloc` functions, write the next procedures

- `addColumn` - Add a new column to the dynamic matrix
- `addRow` - Add a new row to the dynamic matrix
- `read`
  A function which reads from the keyboard: The number of rows, the number of columns, the value to be added in every position. Returns the double-pointer to the matrix

Dynamic Memory Allocation

Let's write some code

Using the `malloc`, `calloc` and `realloc` functions, write the next procedures

- `addColumn` - Add a new column to the dynamic matrix
- `addRow` - Add a new row to the dynamic matrix
- `read`
  A function which reads from the keyboard: The number of rows, the number of columns, the value to be added in every position. Returns the double-pointer to the matrix
- `print`
  A procedure which receives a double pointer, the rows and columns and prints in the screen all the values of the matrix

Outline

Dynamic Memory Allocation - Structs

Matrix

- We can also allocate structures with dynamic memory

Dynamic Memory Allocation - Structs

Matrix

- We can also allocate structures with dynamic memory
- We only need to allocate the necessary space required by the structure

Dynamic Memory Allocation - Structs

Test the code

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

struct Person {
    char name[100];
    char lastName[100];
    int age;
};

int main()
{
    struct Person* myPerson = malloc(sizeof(struct Person));
    myPerson->age = 20;
    printf("Age = %d", myPerson->age);
}
```

Dynamic Memory Allocation - Structs

```
struct Person* myPerson = malloc(sizeof(struct Person));
myPerson->age = 20;
printf("Age = %d", myPerson->age);
```

Dynamic structs

- `struct Person* myPerson`
  A pointer to a Person struct

Dynamic Memory Allocation - Structs

```
struct Person* myPerson = malloc(sizeof(struct Person));
myPerson->age = 20;
printf("Age = %d", myPerson->age);
```

Dynamic structs

- `struct Person* myPerson`
  A pointer to a `Person` struct
- `malloc(sizeof(struct Person))`
  Memory allocation. Get the enough memory space to store a `Person`

Dynamic Memory Allocation - Structs

```
struct Person* myPerson = malloc(sizeof(struct Person));
myPerson->age = 20;
printf("Age = %d", myPerson->age);
```

Dynamic structs

- `struct Person* myPerson`
  A pointer to a `Person` struct
- `malloc(sizeof(struct Person))`
  Memory allocation. Get the enough memory space to store a `Person`
- `myPerson->age`
  We use now the `->` operator to access a member of the structure

Outline

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*
- Functions passed as an argument are widely known as *Callbacks*

Function Pointers

Basics

- Is a characteristic of the *Third Generation* of programming languages ▸ Read More!
- The pointer points to executable code within memory
- The function can be executed just by calling the pointer as a normal function
- Invocation is known as *Indirect Call*
- Functions passed as an argument are widely known as *Callbacks*
- The function is expected to be called back at some convenient time

Function Pointers

```
int myFunction(double a, char b)
```

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` → `int myFunction`
  The pointer type has to be equal to the function return type

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` → `int myFunction`
  The pointer type has to be equal to the function return type
- `(double, char)` → `(double a, double b)`
  Types of the function members

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` → `int myFunction`
  The pointer type has to be equal to the function return type
- `(double, char)` → `(double a, double b)`
  Types of the function members
- `int (*pointer) (double, char) = myFunction`
  The pointer points to `myFunction`

Function Pointers

```
int myFunction(double a, char b)
int (*pointer) (double, char)
```

Declaring a pointer

- `int (*pointer)` → `int myFunction`
  The pointer type has to be equal to the function return type
- `(double, char)` → `(double a, double b)`
  Types of the function members
- `int (*pointer) (double, char) = myFunction`
  The pointer points to `myFunction`
- `void process(void (*funcp)(int), int a, int b)`
  A function which receives a function as a parameter

Function Pointers

Test the code

```c
#include <stdio.h>

void callback(int value)
{
    printf("This Callback function prints the value = %i", value);
}
void process(void (*funcp)(int), int a, int b)
{
    int c = a + b;
    funcp(c);
}
int main()
{
    void (*functionPointer)(int);
    functionPointer = callback;
    process(functionPointer, 2, 3);
}
```

Function Pointers

Test the code

```
#include <stdio.h>
void callback()
{
    printf("The process has finished!!!\n");
}
void process(void (*funcp)(), int *a, int b)
{
    *a = (*a * *a) + b;
    funcp();
}
int main()
{
    int var = 5;
    void (*functionPointer)();
    functionPointer = callback;
    process(functionPointer, &var, 3);
    printf("a = %d", var);
}
```

Function Pointers

Test the code

```c
#include <stdio.h>
int sumCallback(int a, int b)
{
    return a + b;
}
void process(int (*funcp)(int, int), int a, int b)
{
    printf("a = %d, b = %d, res = %d", a, b, funcp(a, b));
}
int main()
{
    int (*functionPointer)(int, int);
    functionPointer = sumCallback;
    process(functionPointer, 5, 3);
}
```

Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

The map function

- Is a *High-order* function

Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

The map function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments

Function Pointers

```
int* map(int (*fun)(int value), int *array, int size)
```

The `map` function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments
- The `map` function applies the input function over all the elements in an array

Function Pointers

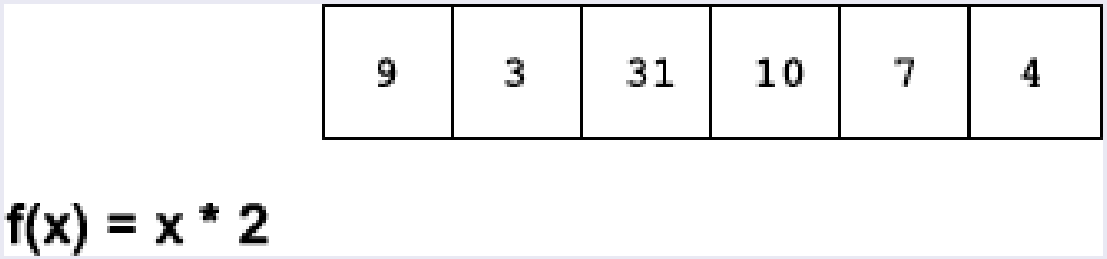
```
int* map(int (*fun)(int value), int *array, int size)
```

The `map` function

- Is a *High-order* function
- A high-order function is the one who takes functions as arguments
- The `map` function applies the input function over all the elements in an array
- *Keep in mind:* It works also for sequential *containers*

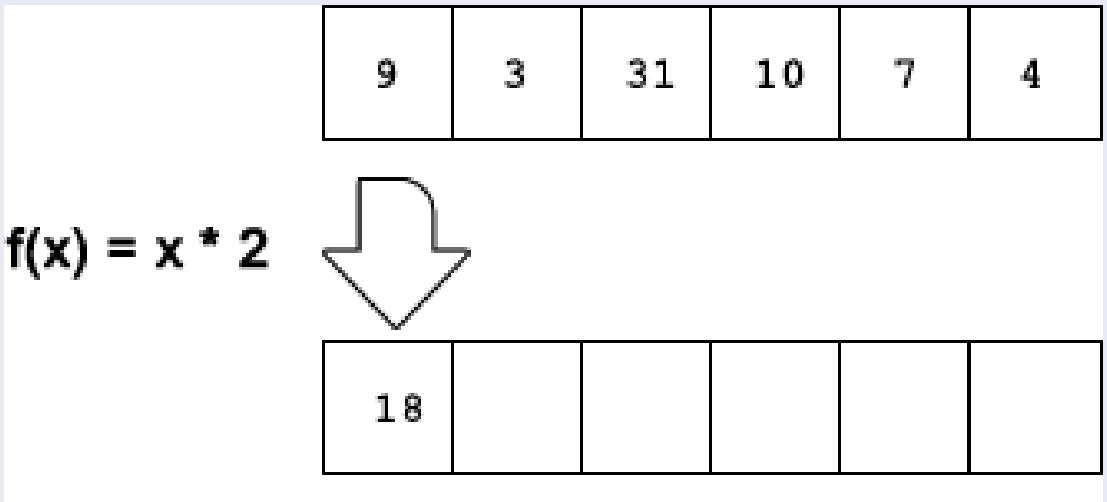
Function Pointers

The map function - How it works?



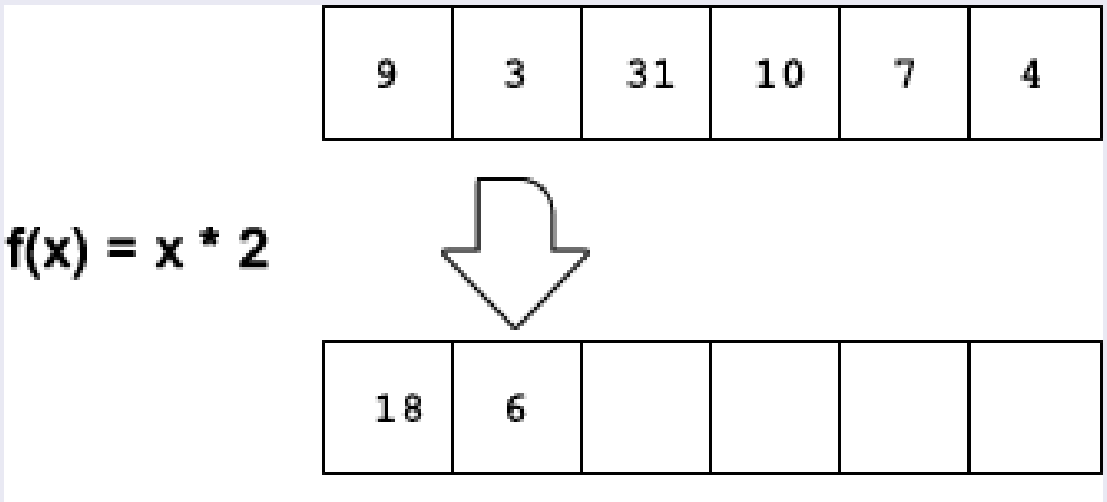
Function Pointers

The map function - How it works?



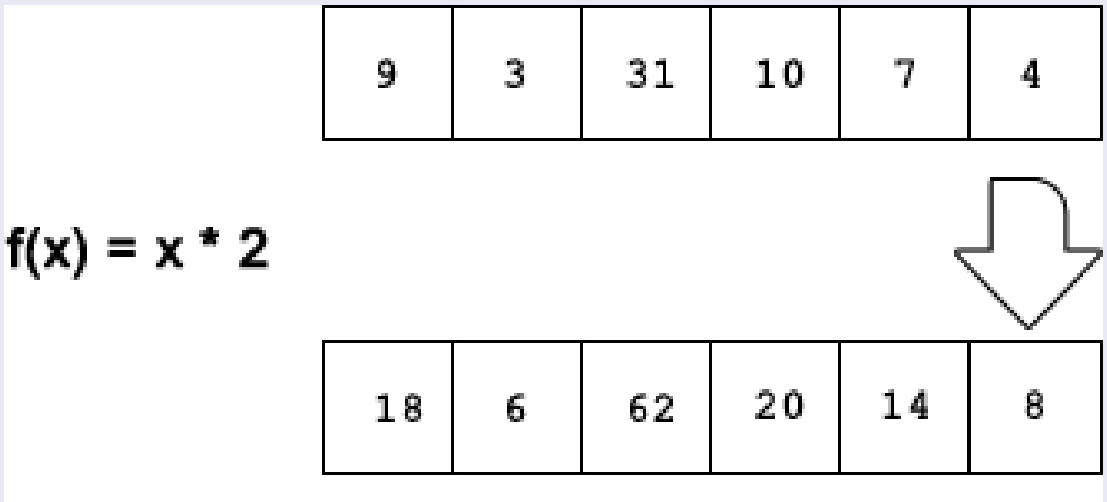
Function Pointers

The map function - How it works?



Function Pointers

The map function - How it works?



Function Pointers

The map function - Implementation

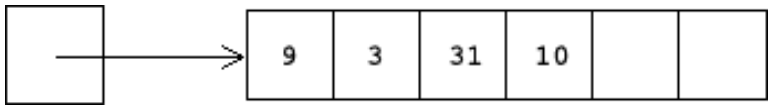
```
int* map(int (*funPtr)(int), int* ptr, int size)
{
    int* ptrRes = calloc(size, sizeof(int));
    for (int i=0; i<size; i++)
    {
        *(ptrRes+i) = funPtr(ptr[i]);
    }
    return ptrRes;
}
```

Function Pointers - Test the code with the map function

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

int getDouble(int a)
{
    return a*2;
}

int main()
{
    int *a = calloc(2, sizeof(int));
    int (*fun)(int) = getDouble;
    a[0] = 5;
    a[1] = 9;

    int* array = map(fun, a,2);

    printf("%d, %d", array[0], array[1]);
}
```

Outline

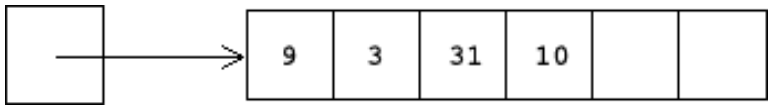
Array List



Definition

An Array List is the most simple definition of a dynamic array.

Array List

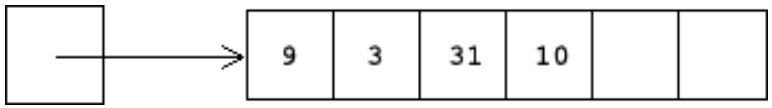


Definition

An Array List is the most simple definition of a dynamic array.

- Has a fixed size, but has the ability to be expanded or contract when is required

Array List

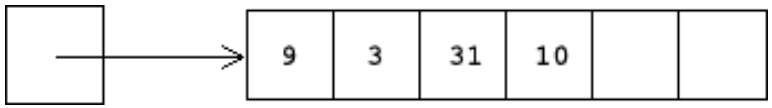


Definition

An Array List is the most simple definition of a dynamic array.

- Has a fixed size, but has the ability to be expanded or contract when is required
- Programmer don't care about to reallocate memory

Array List

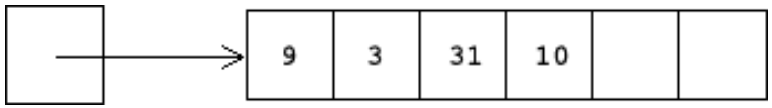


Definition

An Array List is the most simple definition of a dynamic array.

- Has a fixed size, but has the ability to be expanded or contract when is required
- Programmer don't care about to reallocate memory
- Easy to implement and use

Array List

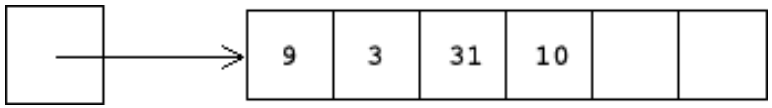


Definition

An Array List is the most simple definition of a dynamic array.

- Has a fixed size, but has the ability to be expanded or contract when is required
- Programmer don't care about to reallocate memory
- Easy to implement and use

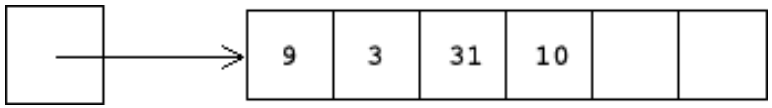
Array List



Definition

- Values are contiguous in memory

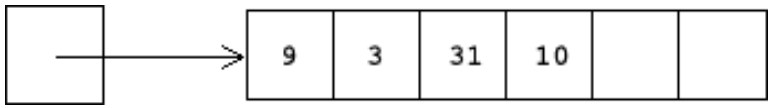
Array List



Definition

- Values are contiguous in memory
- We have to iterate over the whole list to get a value

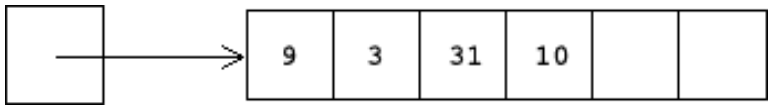
Array List



Definition

- Values are contiguous in memory
- We have to iterate over the whole list to get a value
- Values are added at the end of the array list

Array List



Definition

- Values are contiguous in memory
- We have to iterate over the whole list to get a value
- Values are added at the end of the array list
- Delete values operation is a high complexity job!

Array List

Operations

- `insert`
  Input: A pointer to the array list and the element to insert
  Do: Inserts the element at the last position of the list

Array List

Operations

- `insert`
  Input: A pointer to the array list and the element to insert
  Do: Inserts the element at the last position of the list
- `delete`
  Input: A pointer to the array list and the element to delete
  Do: Search for the element, and delete the element

Array List

Operations

- `insert`
  Input: A pointer to the array list and the element to insert
  Do: Inserts the element at the last position of the list
- `delete`
  Input: A pointer to the array list and the element to delete
  Do: Search for the element, and delete the element
- `valueAt`
  Input: A pointer to the array list and the position to retrieve
  Do: Search for the element, and return the value at the position

Outline

Array List - Building

The `ArrayList` struct

Contains the next fields:

- `int capacity`
  The capacity of the Array List

Array List - Building

The `ArrayList` struct

Contains the next fields:

- `int capacity`
  The capacity of the Array List
- `int elements`
  Number of elements in the Array List

Array List - Building

The `ArrayList` struct

Contains the next fields:

- `int capacity`
  The capacity of the Array List
- `int elements`
  Number of elements in the Array List
- `int increment`
  The number of positions that the array list will expand when adding a value exceeds the capacity

Array List - Building

The `ArrayList` struct

Contains the next fields:

- `int capacity`
  The capacity of the Array List
- `int elements`
  Number of elements in the Array List
- `int increment`
  The number of positions that the array list will expand when adding a value exceeds the capacity
- `int* basicArray`
  The number of positions that the array list will expand when adding a value exceeds the capacity

Array List - Insert

Algorithm

1. Let `pointer` a pointer to the Array List structure, and `data` an integer value to be inserted

Array List - Insert

Algorithm

1. Let `pointer` a pointer to the Array List structure, and `data` an integer value to be inserted
2. Check if the number of elements in the array is greater than the Array List capacity

Array List - Insert

Algorithm

1. Let `pointer` a pointer to the Array List structure, and `data` an integer value to be inserted
2. Check if the number of elements in the array is greater than the Array List capacity
3. If so, we have to *EXPAND* the array list. Otherwise, no

Array List - Insert

Algorithm

1. Let `pointer` a pointer to the Array List structure, and `data` an integer value to be inserted
2. Check if the number of elements in the array is greater than the Array List capacity
3. If so, we have to *EXPAND* the array list. Otherwise, no
4. Insert the element at the end of the `basicArray`

Array List - Insert

Algorithm

1. Let `pointer` a pointer to the Array List structure, and `data` an integer value to be inserted
2. Check if the number of elements in the array is greater than the Array List capacity
3. If so, we have to *EXPAND* the array list. Otherwise, no
4. Insert the element at the end of the `basicArray`
5. Increase the value of the `elements` field

Array List - Expand

Algorithm

To expand the size of the array list, we just need to reallocate the `basicArray` into a bigger one

Array List - Expand

Algorithm

To expand the size of the array list, we just need to reallocate the `basicArray` into a bigger one

1. If the expand is needed, reallocate (Using the `realloc` function) the `basicArray`, with the new size = `capacity` + `increment`

Array List - Expand

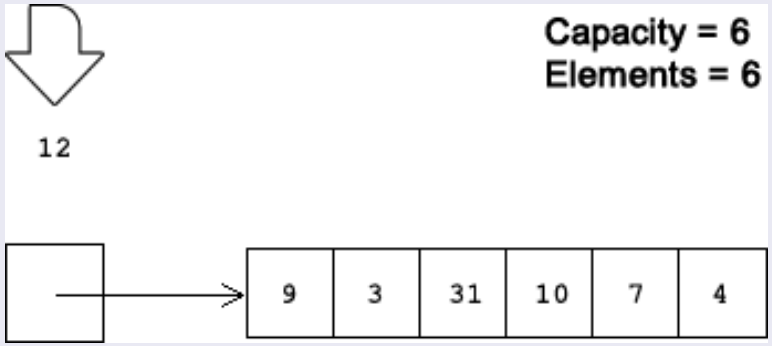
Algorithm

To expand the size of the array list, we just need to reallocate the `basicArray` into a bigger one

1. If the expand is needed, reallocate (Using the `realloc` function) the `basicArray`, with the new size = `capacity` + `increment`
2. Update the `capacity` field

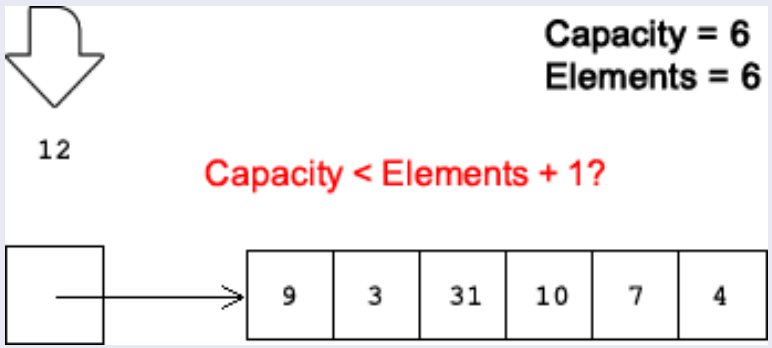
Array List - Insert

Insert an element - Example



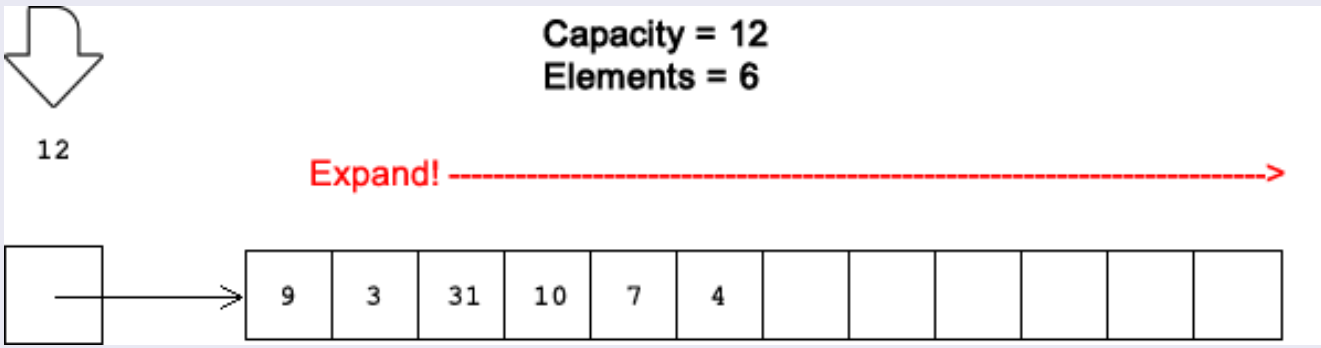
Array List - Insert

Insert an element - Example



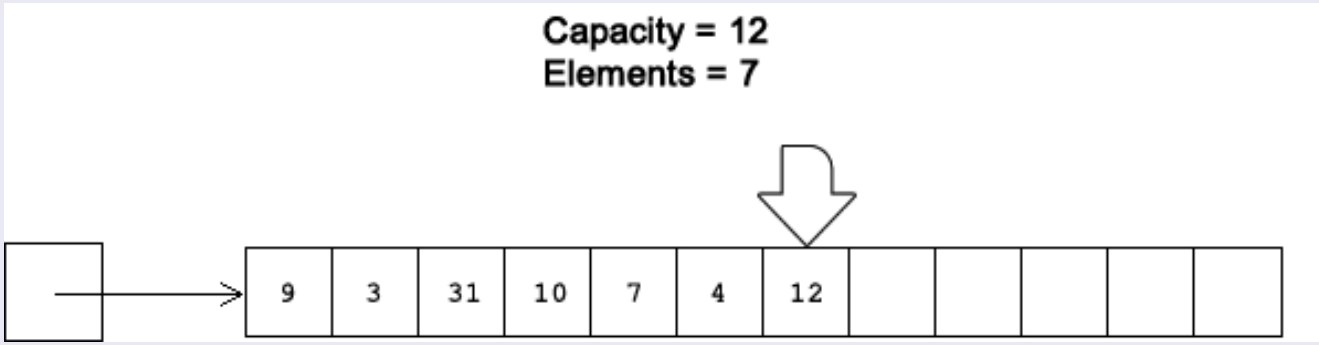
Array List - Insert

Insert an element - Example



Array List - Insert

Insert an element - Example



Array List - Delete

Algorithm

1. Let `pointer` a pointer to the array list structure, and `data` an integer value to be deleted

Array List - Delete

Algorithm

1. Let `pointer` a pointer to the array list structure, and `data` an integer value to be deleted
2. Search the value to delete into the `basicArray`

Array List - Delete

Algorithm

1. Let `pointer` a pointer to the array list structure, and `data` an integer value to be deleted
2. Search the value to delete into the `basicArray`
3. If the value was found, store the position index were it was found, otherwise, stop the algorithm

Array List - Delete

Algorithm

1. Let `pointer` a pointer to the array list structure, and `data` an integer value to be deleted
2. Search the value to delete into the `basicArray`
3. If the value was found, store the position index were it was found, otherwise, stop the algorithm
4. Decrease the number of `elements`

Array List - Delete

Algorithm

1. Let `pointer` a pointer to the array list structure, and `data` an integer value to be deleted
2. Search the value to delete into the `basicArray`
3. If the value was found, store the position index were it was found, otherwise, stop the algorithm
4. Decrease the number of `elements`
5. Move (with `memmove`) one position to the left, all the elements at the right of the found element

Array List - Delete

Algorithm

1. Let `pointer` a pointer to the array list structure, and `data` an integer value to be deleted
2. Search the value to delete into the `basicArray`
3. If the value was found, store the position index were it was found, otherwise, stop the algorithm
4. Decrease the number of `elements`
5. Move (with `memmove`) one position to the left, all the elements at the right of the found element
6. Check if the array needs to be *CONTRACTED* if so, contract

Array List - Contract

Algorithm

To contract the size of the array list, we just need to reallocate the `basicArray` into a smaller one

Array List - Contract

Algorithm

To contract the size of the array list, we just need to reallocate the `basicArray` into a smaller one

1. Check if the `elements` of the array are less thant the `capacity - increment`

Array List - Contract

Algorithm

To contract the size of the array list, we just need to reallocate the `basicArray` into a smaller one

1. Check if the `elements` of the array are less thant the `capacity - increment`
2. If so, the contraction is needed

Array List - Contract

Algorithm

To contract the size of the array list, we just need to reallocate the `basicArray` into a smaller one

1. Check if the `elements` of the array are less thant the `capacity - increment`
2. If so, the contraction is needed
3. Reallocate (Using the `realloc` function) the `basicArray`, with the new size = `capacity - increment`

Array List - Contract

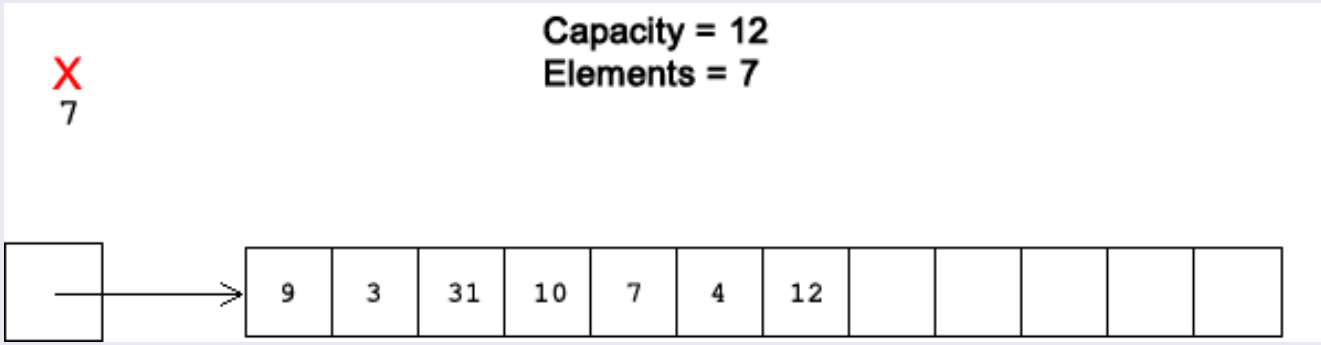
Algorithm

To contract the size of the array list, we just need to reallocate the `basicArray` into a smaller one

1. Check if the `elements` of the array are less thant the `capacity - increment`
2. If so, the contraction is needed
3. Reallocate (Using the `realloc` function) the `basicArray`, with the new size = `capacity - increment`
4. Update the `capacity` field

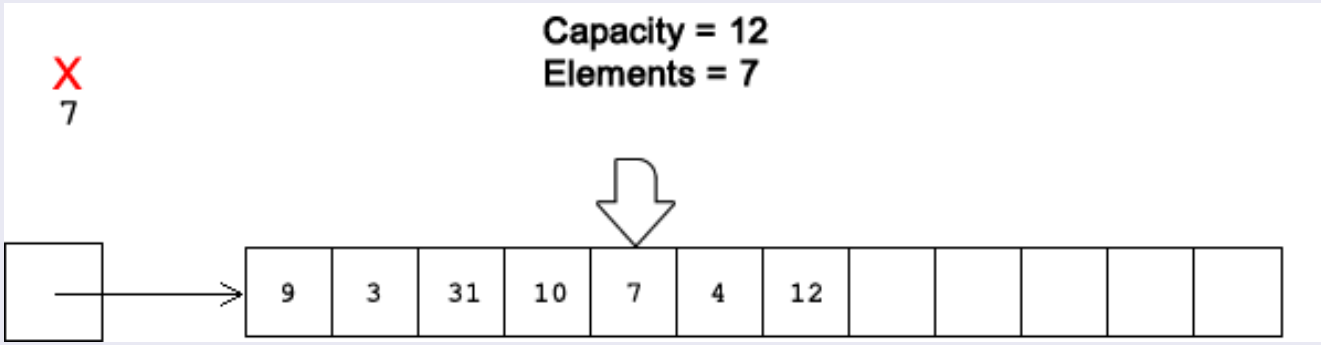
Array List - Delete

Delete an element - Example



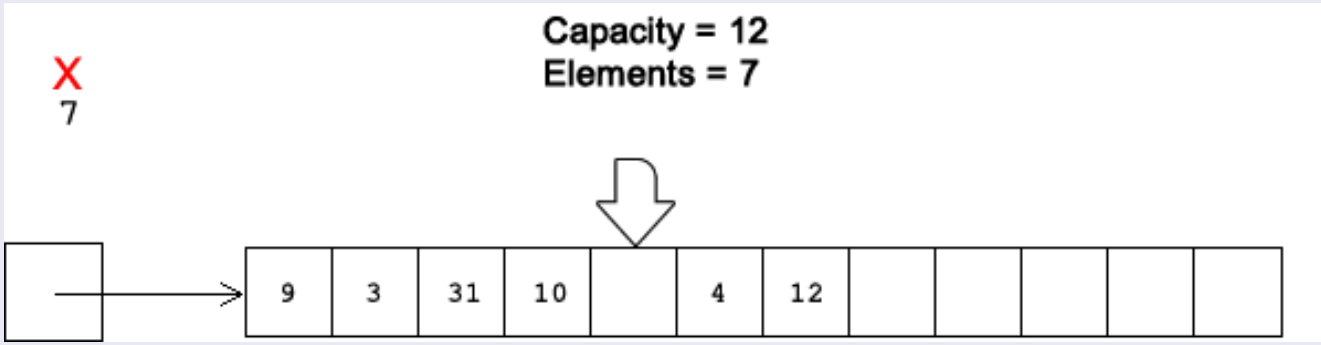
Array List - Delete

Delete an element - Example



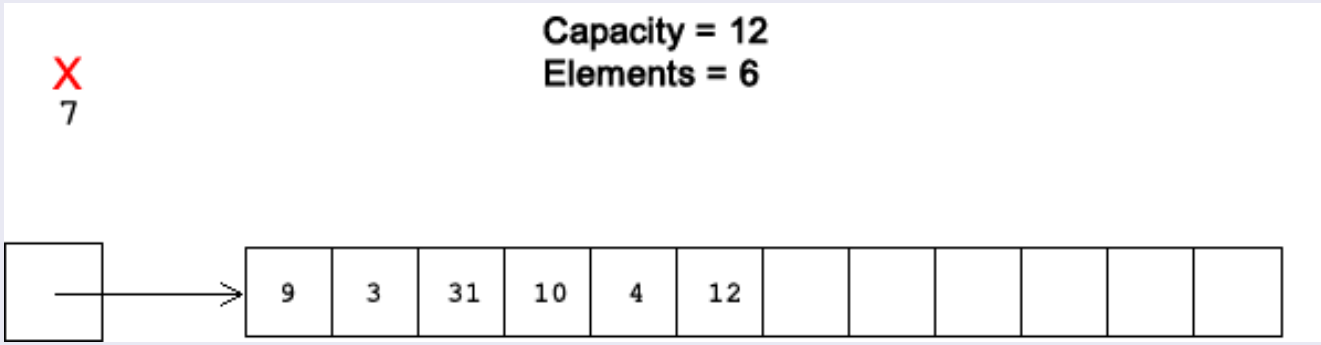
Array List - Delete

Delete an element - Example



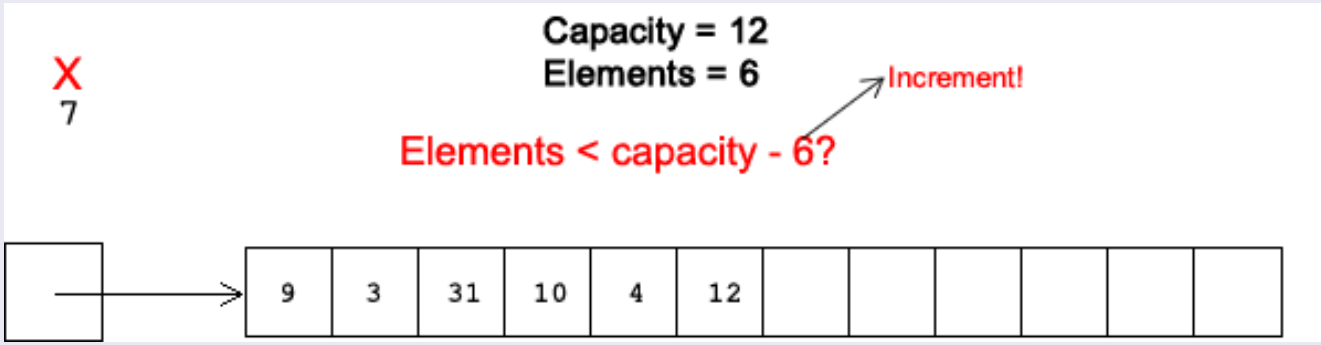
Array List - Delete

Delete an element - Example



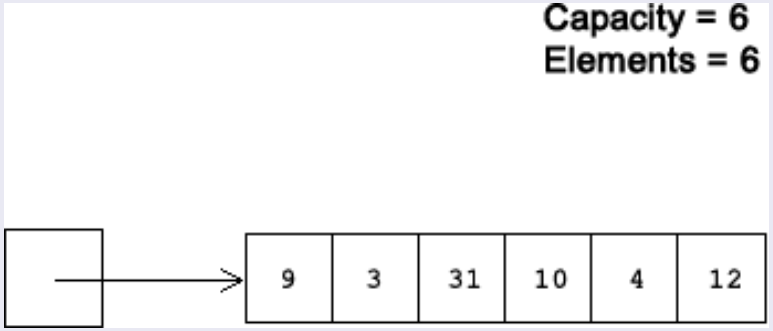
Array List - Delete

Delete an element - Example



Array List - Delete

Delete an element - Example



Array List - Find

ValueAt

1. Let `pointer` a pointer to the Array List structure, and `index` the index to the position to retrieve

Array List - Find

ValueAt

1. Let `pointer` a pointer to the Array List structure, and `index` the index to the position to retrieve
2. Check if the `index` is greater or equal to `0`, or less than the number of `elements` in the array

Array List - Find

ValueAt

1. Let `pointer` a pointer to the Array List structure, and `index` the index to the position to retrieve
2. Check if the `index` is greater or equal to `0`, or less than the number of `elements` in the array
3. If so, return the value at the `index` position in the `basicArray`

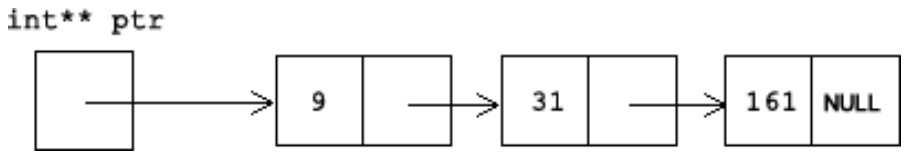
Array List - Find

ValueAt

1. Let `pointer` a pointer to the Array List structure, and `index` the index to the position to retrieve
2. Check if the `index` is greater or equal to `0`, or less than the number of `elements` in the array
3. If so, return the value at the `index` position in the `basicArray`

Outline

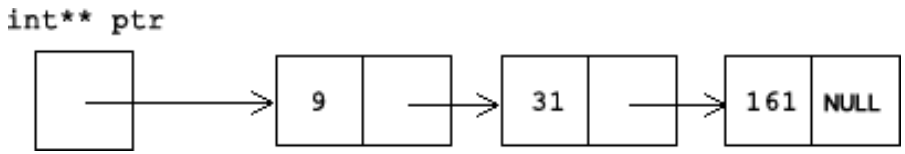
Linked List



Definition

A Linked List is a collection of structures connected by pointers (links).

Linked List

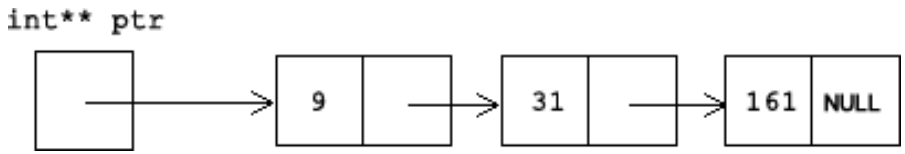


Definition

A Linked List is a collection of structures connected by pointers (links).

- Every *node* of the list contains two elements: The value and a pointer to the next node

Linked List

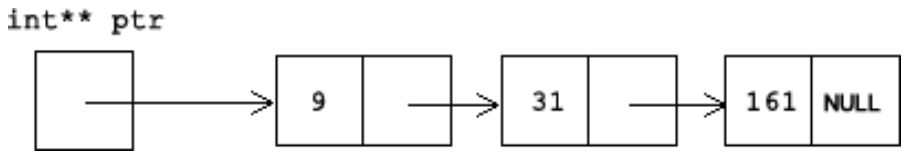


Definition

A Linked List is a collection of structures connected by pointers (links).

- Every *node* of the list contains two elements: The value and a pointer to the next node
- The end of the list is the pointer with `NULL`

Linked List

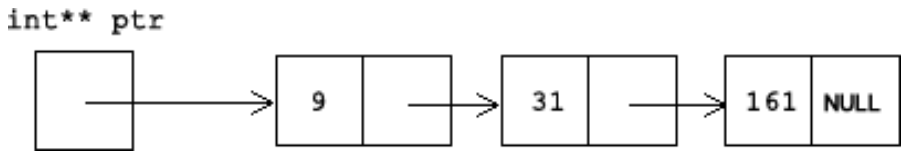


Definition

A Linked List is a collection of structures connected by pointers (links).

- Every *node* of the list contains two elements: The value and a pointer to the next node
- The end of the list is the pointer with `NULL`
- Reduces the complexity of the array list when we are adding and removing elements

Linked List

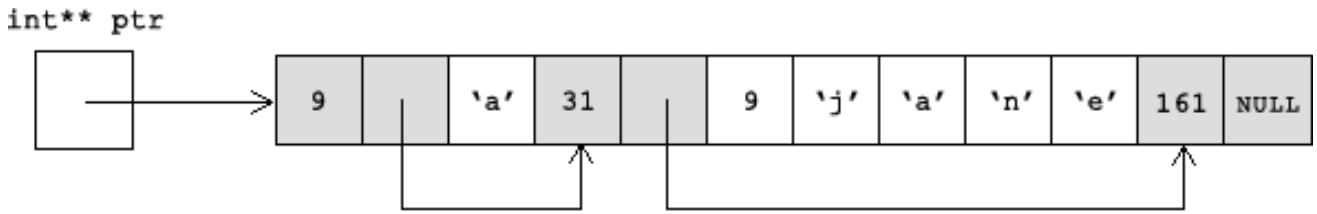


Definition

A Linked List is a collection of structures connected by pointers (links).

- Every *node* of the list contains two elements: The value and a pointer to the next node
- The end of the list is the pointer with `NULL`
- Reduces the complexity of the array list when we are adding and removing elements

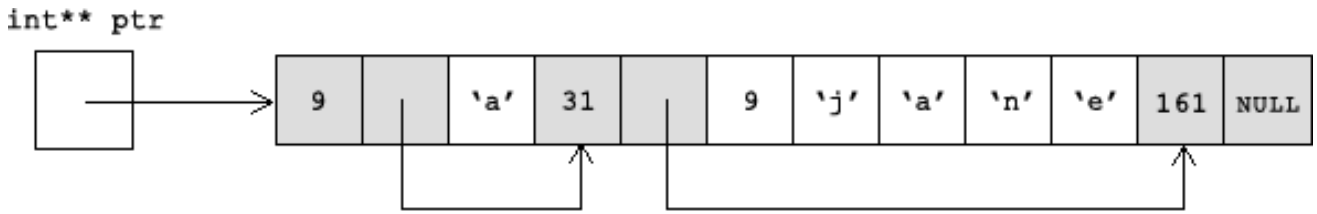
Linked List



Definition

- Values are not contiguous in memory

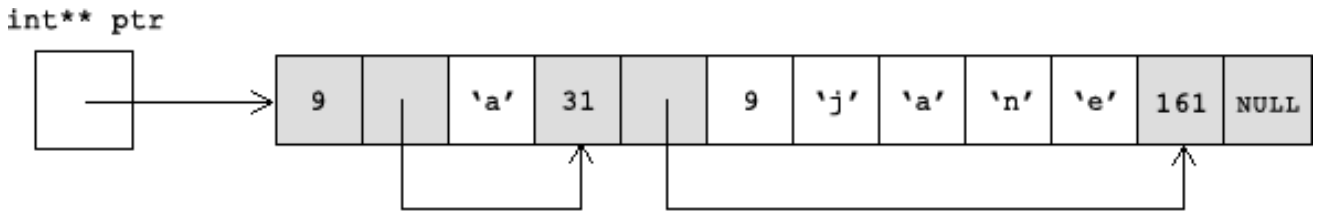
Linked List



Definition

- Values are not contiguous in memory
- We have to iterate over the whole list to get a value

Linked List



Definition

- Values are not contiguous in memory
- We have to iterate over the whole list to get a value
- To insert a value is just enough to create a new node and change the involved *links*

Linked List

Operations

- `insert`
  Input: A pointer to the linked list and the element to insert
  Do: Inserts the element at the last position of the list

Linked List

Operations

- `insert`
  Input: A pointer to the linked list and the element to insert
  Do: Inserts the element at the last position of the list
- `delete`
  Input: A pointer to the linked list and the element to delete
  Do: Search for the element, and delete the node

Linked List

Operations

- `insert`
  Input: A pointer to the linked list and the element to insert
  Do: Inserts the element at the last position of the list
- `delete`
  Input: A pointer to the linked list and the element to delete
  Do: Search for the element, and delete the node
- `find`
  Input: A pointer to the linked list and the element to search
  Do: Search for the element node per node, and return `1` if the element was found, otherwise `0`

Outline

Linked List - Building

The `Node` struct

Contains the next fields:

- `int value`
  Where the value will be stored
- `Node* next`
  A pointer to the next node

Linked List - Insert

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be inserted

Linked List - Insert

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be inserted
2. Check if the pointer to the `next` position is null, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching

Linked List - Insert

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be inserted
2. Check if the pointer to the `next` position is null, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. Allocate a new `Node` in the pointer to `next`

Linked List - Insert

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be inserted
2. Check if the pointer to the `next` position is null, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. Allocate a new `Node` in the pointer to `next`
4. Assign the value to the node in `next`

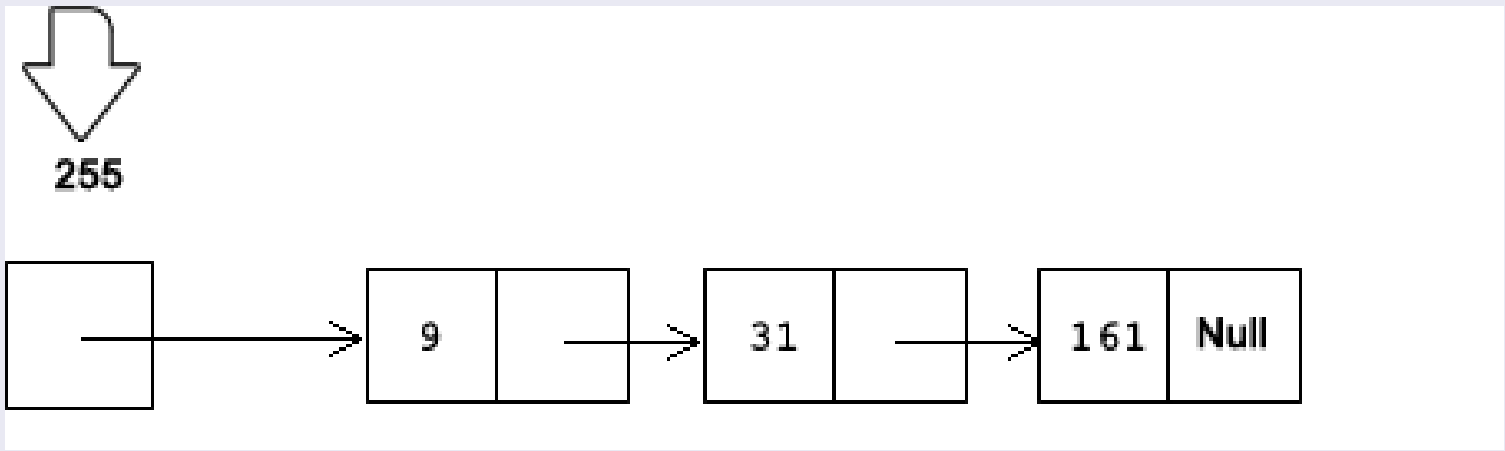
Linked List - Insert

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be inserted
2. Check if the pointer to the `next` position is null, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. Allocate a new `Node` in the pointer to `next`
4. Assign the value to the node in `next`
5. Points to null the `next` pointer of the `next` node

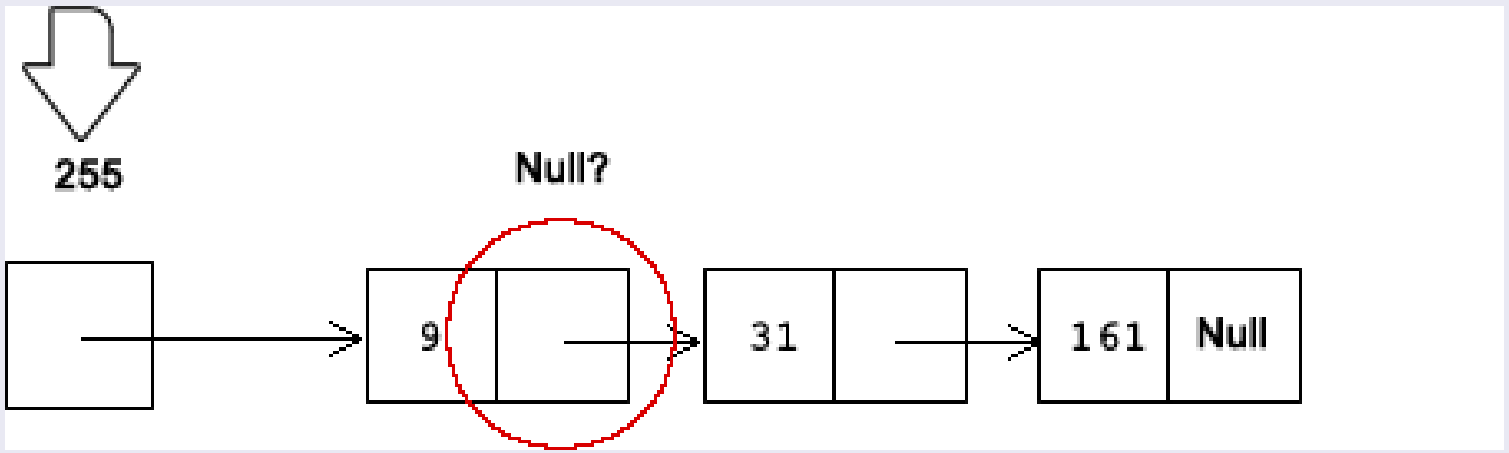
Linked List - Insert

Insert a node - Example



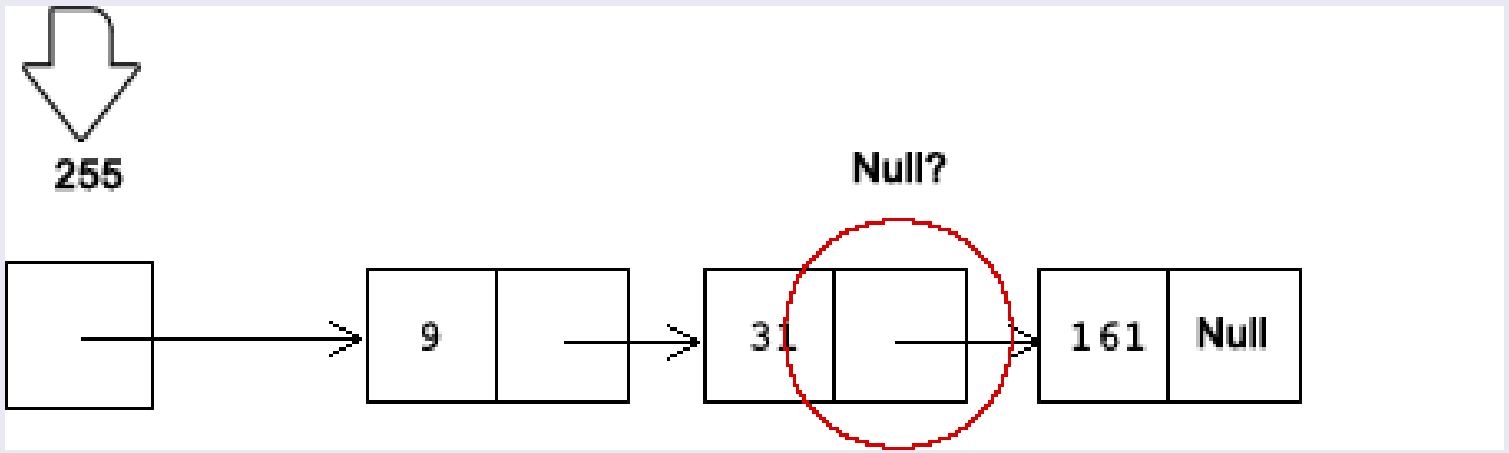
Linked List - Insert

Insert a node - Example



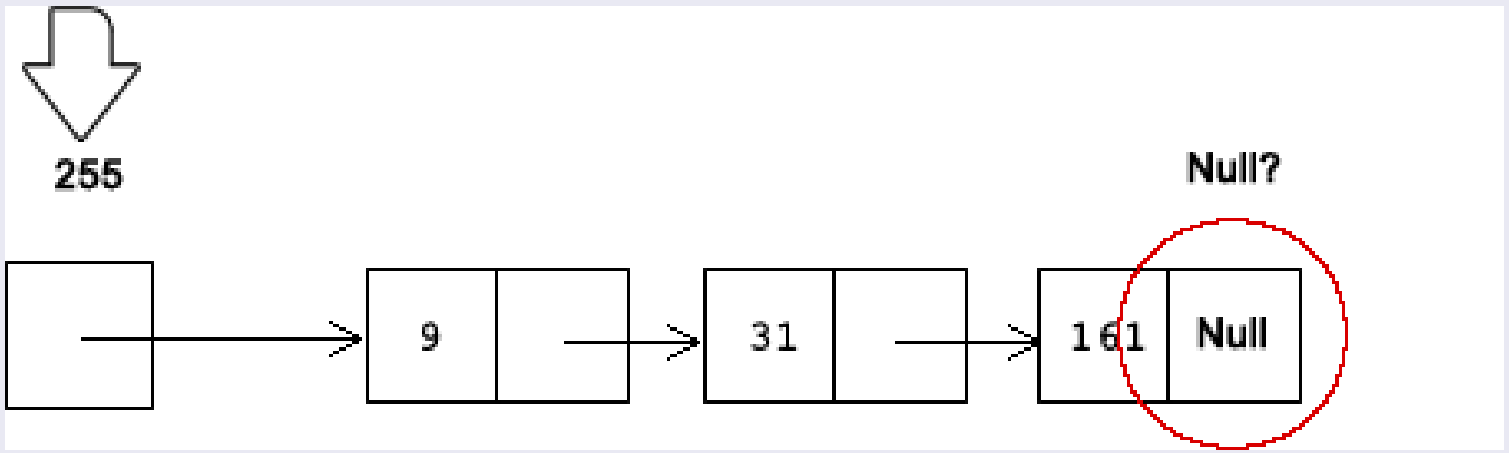
Linked List - Insert

Insert a node - Example



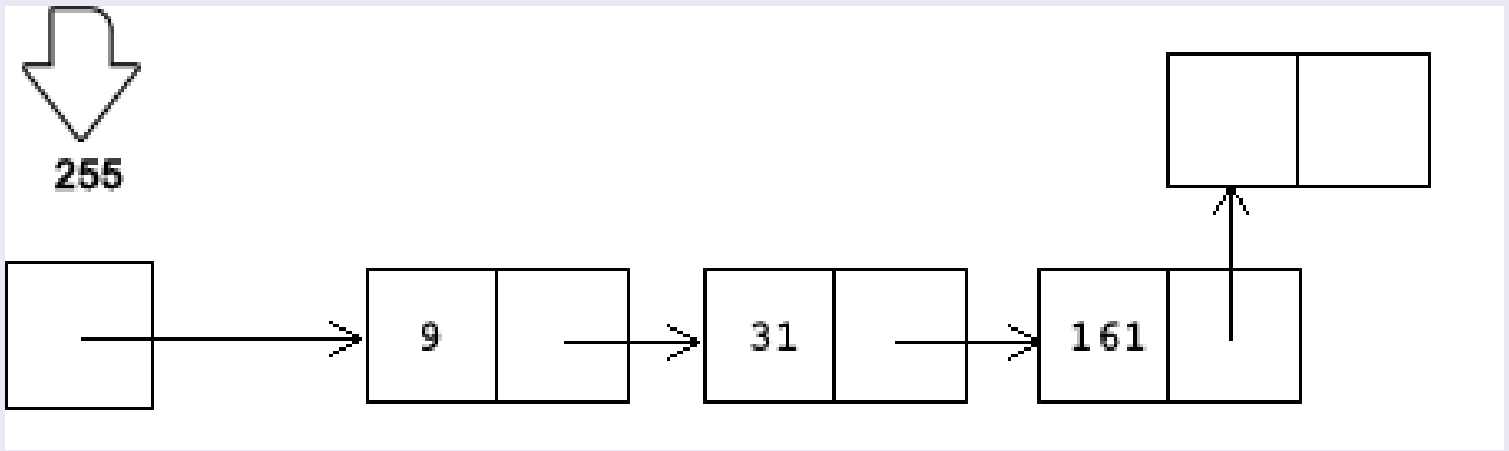
Linked List - Insert

Insert a node - Example



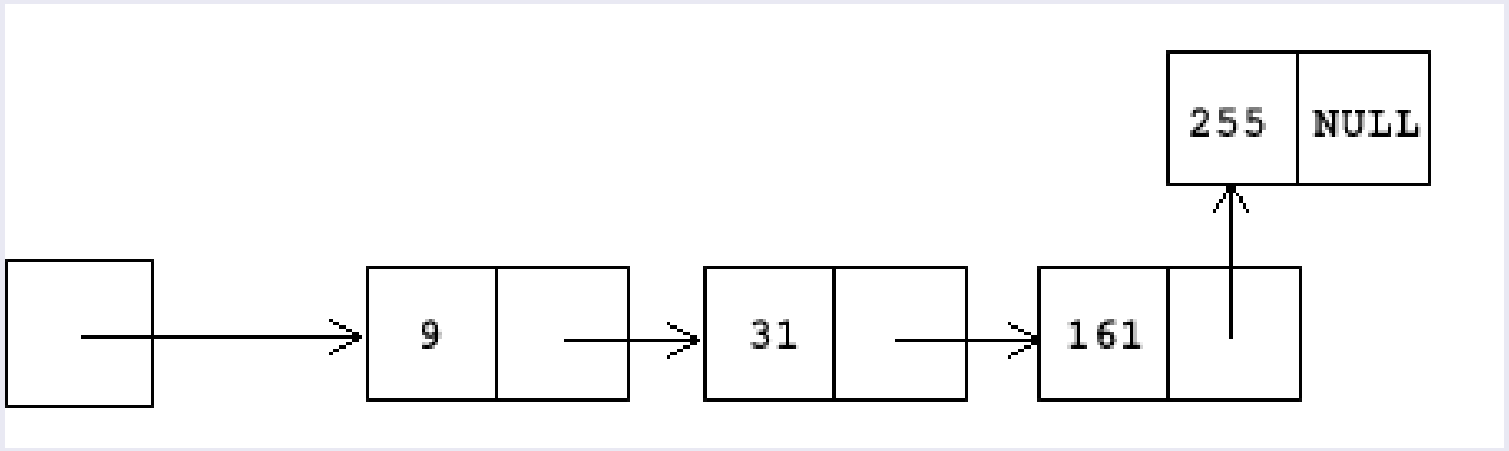
Linked List - Insert

Insert a node - Example



Linked List - Insert

Insert a node - Example



Linked List - Delete

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be deleted

Linked List - Delete

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be deleted
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching

Linked List - Delete

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be deleted
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. If the pointer to the next position is null, the value has not been found so, print an error message and stop

Linked List - Delete

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be deleted
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. If the pointer to the next position is null, the value has not been found so, print an error message and stop
4. If the data has been found in the `next` node, store the reference to the `next` node in a `temp` pointer

Linked List - Delete

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be deleted
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. If the pointer to the next position is null, the value has not been found so, print an error message and stop
4. If the data has been found in the `next` node, store the reference to the `next` node in a `temp` pointer
5. Points the `next` field of the current node, to the `next` field of the `temp` node

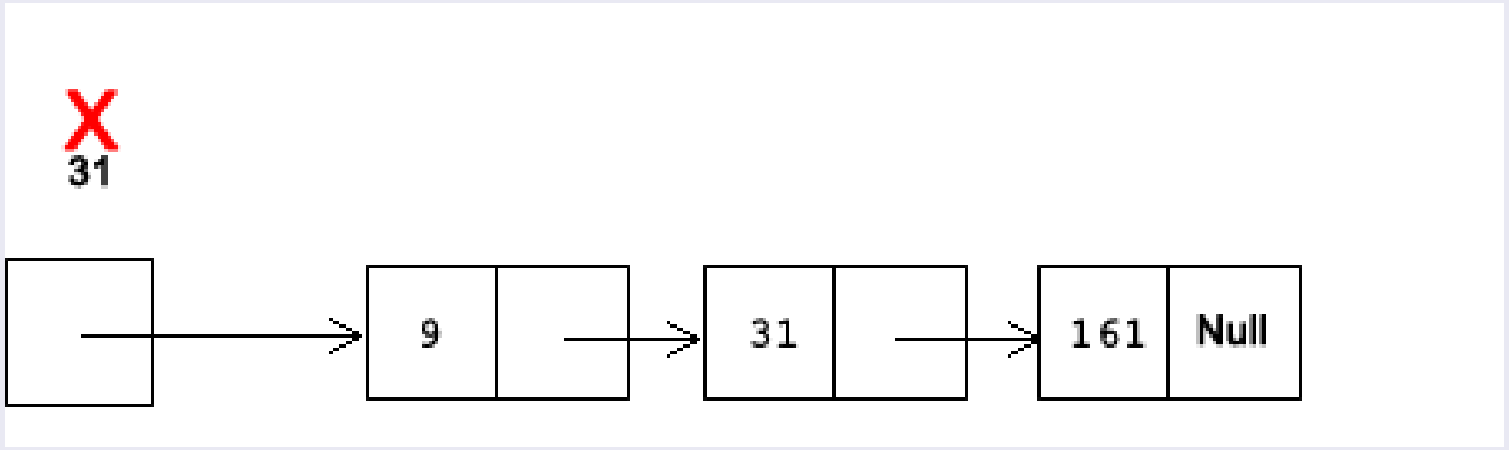
Linked List - Delete

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be deleted
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. If the pointer to the next position is null, the value has not been found so, print an error message and stop
4. If the data has been found in the `next` node, store the reference to the `next` node in a `temp` pointer
5. Points the `next` field of the current node, to the `next` field of the `temp` node

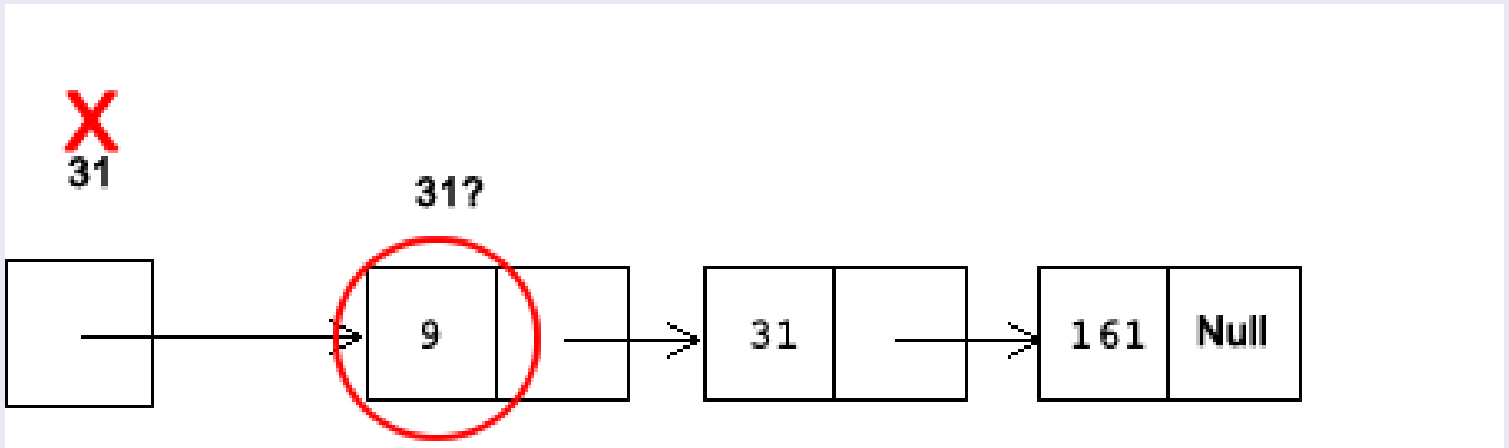
Linked List - Delete

Delete a node - Example



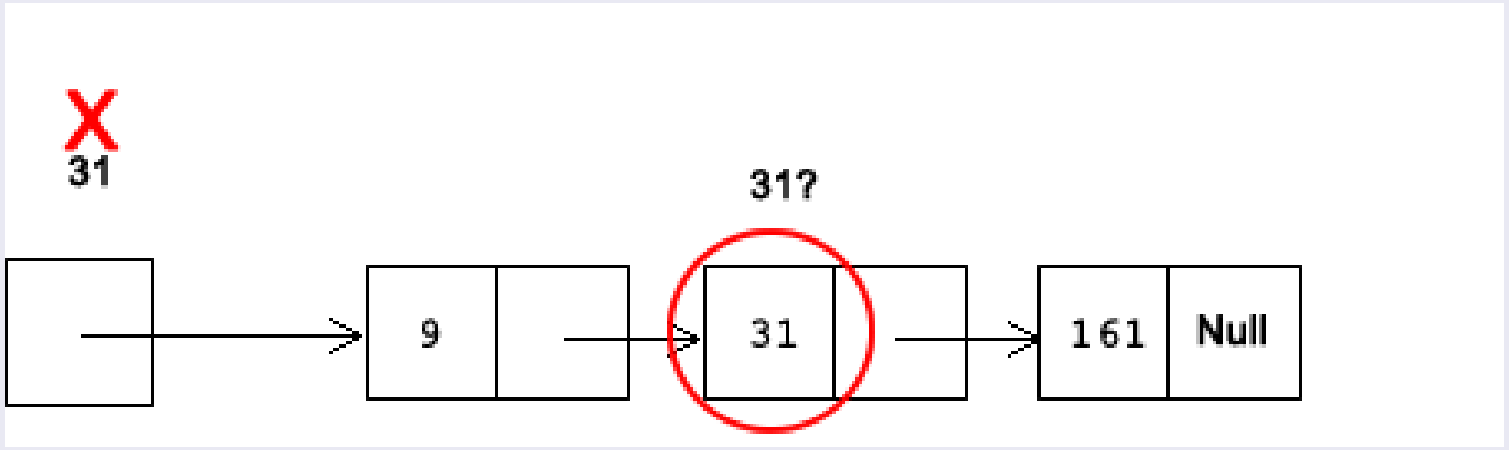
Linked List - Delete

Delete a node - Example



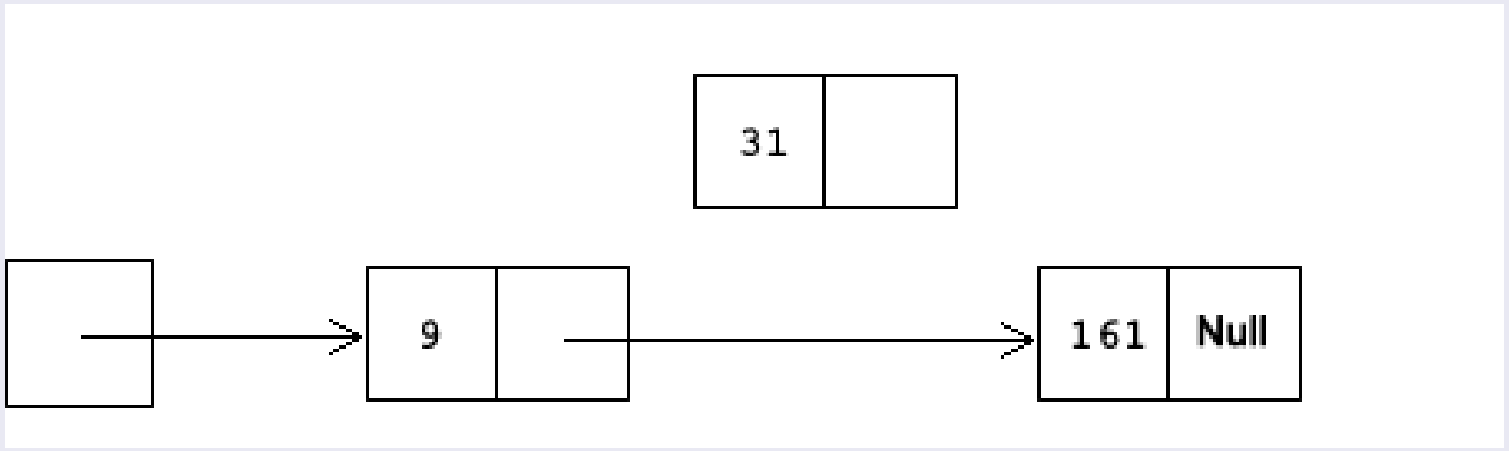
Linked List - Delete

Delete a node - Example



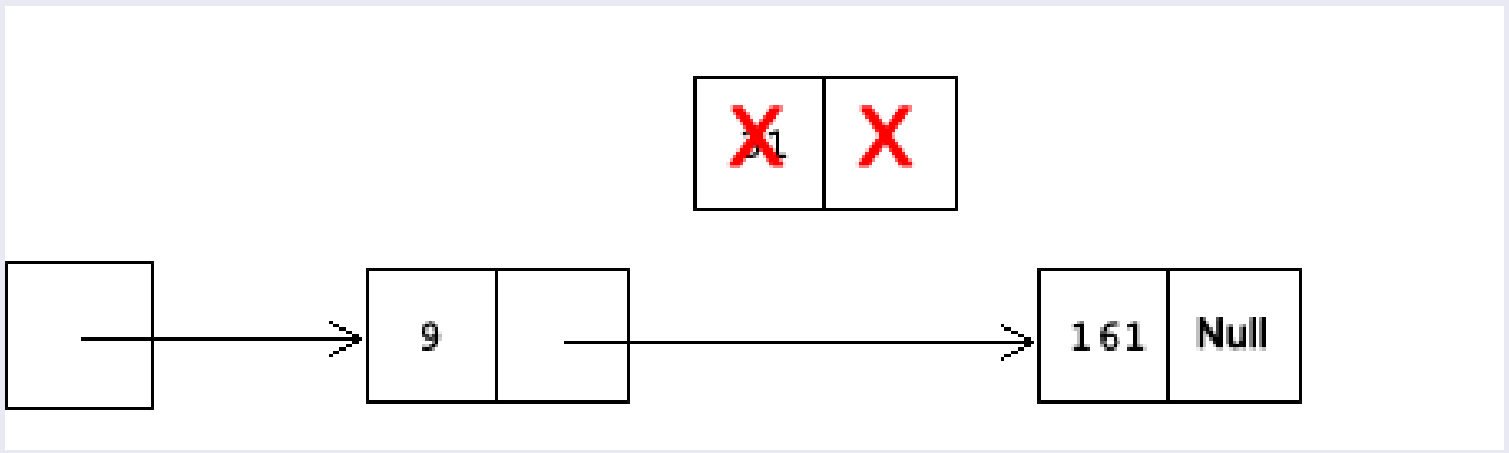
Linked List - Delete

Delete a node - Example



Linked List - Delete

Delete a node - Example



Linked List - Find

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be find

Linked List - Find

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be find
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching

Linked List - Find

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be find
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. If the pointer to the next position is null, the value has not been found so, return 0

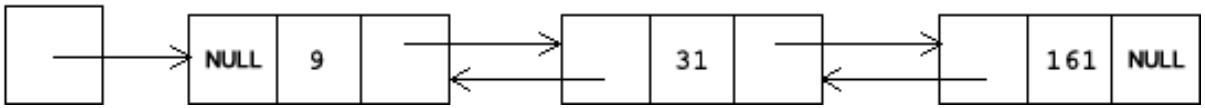
Linked List - Find

Algorithm

1. Let `pointer` a pointer to the head node of the Linked List, and `data` an integer value to be find
2. Check if the pointer to the `next` position is null and if the data of the pointer to the next position is what I'm looking for, if so, check the next step, otherwise move the pointer to the `next` reference and keep it searching
3. If the pointer to the next position is null, the value has not been found so, return 0
4. If the data has been found in the `next` node, return 1

Outline

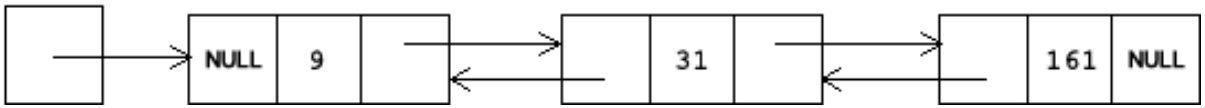
Doubly Linked List



Definition

A Doubly Linked List is a new abstraction of a Linked list where we can traverse the list from back to front

Doubly Linked List

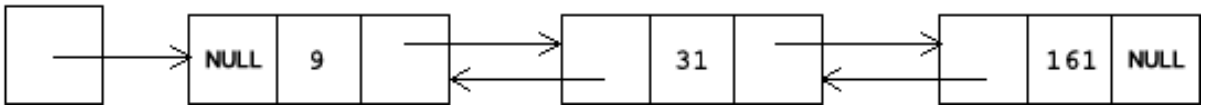


Definition

A Doubly Linked List is a new abstraction of a Linked list where we can traverse the list from back to front

- Every *node* of the list contains tree elements:
  - The stored value

Doubly Linked List

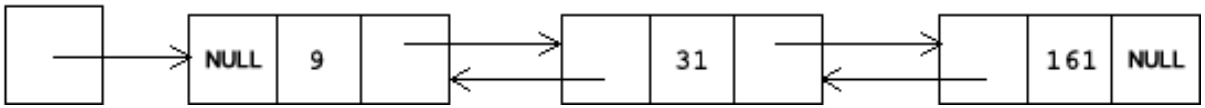


Definition

A Doubly Linked List is a new abstraction of a Linked list where we can traverse the list from back to front

- Every *node* of the list contains tree elements:
  - The stored value
  - A pointer to the next node

Doubly Linked List



Definition

A Doubly Linked List is a new abstraction of a Linked list where we can traverse the list from back to front

- Every *node* of the list contains tree elements:
  - The stored value
  - A pointer to the next node
  - A pointer to the previous node

Linked List

Operations

- `insert`
  Input: A pointer to the linked list and the element to insert
  Do: Inserts the element at the last position of the list

Linked List

Operations

- `insert`
  Input: A pointer to the linked list and the element to insert
  Do: Inserts the element at the last position of the list
- `delete`
  Input: A pointer to the linked list and the element to delete
  Do: Search for the element, and delete the node

Linked List

Operations

- `insert`
  Input: A pointer to the linked list and the element to insert
  Do: Inserts the element at the last position of the list
- `delete`
  Input: A pointer to the linked list and the element to delete
  Do: Search for the element, and delete the node
- `find`
  Input: A pointer to the linked list and the element to search
  Do: Search for the element node per node, and return `1` if the element was found, otherwise `0`