Fundamental Data Structures & Algorithms using the C language

David Corzo

2020 May 18

Contents

1 Linked List **4**

1.1 Introduction to linked lists 4

1.1.1 What is wrong with arrays? 4

1.2 Definition of linked list, conception of node, understanding basic principles 5

1.2.1 Example for storing a set of integers 5

1.3 Categories of linked lists - singly, doubly and circular linked list 5

1.3.1 Singly linked lists 5

1.3.2 Circular linked list 5

1.3.3 Double linked list 6

2 Singly linked lists **7**

2.1 Linked list operation for insert at tail 8

2.2 Linked list operation for inserting at head 9

2.3 Linked list operation for traversing singly linked lists 10

2.4 Linked list operation for delete first 11

2.5 Linked list operation for delete last 11

2.6 Linked list operation for deleting target nodes 11

2.7 Linked list operation for finding a node 11

2.8 Linked list operation reverse a singly linked list 11

2.9 Linked list operation for printing and traversing the linked list recursively 13

3 Doubly linked list **14**

3.1 Introduction to doubly linked list 14

3.2 Doubly linked list operation for adding node at head 14

3.3 Doubly linked list operation for adding node at tail 14

3.4 Doubly linked list operation for finding a node in the list 14

3.5 Doubly linked list operation for deleting node at head 14

3.6 Doubly linked list operation for deleting node at tail 14

3.7 Doubly linked list operation for deleting a target node 14

3.8 Doubly linked list for traversing and printing data of the doubly linked list 14

4 Circular linked lists **15**

4.1 Introduction 15

4.1.1 Circular linked list visualization 15

4.2 Operation for inserting a node to circular linked list 17

4.3 Operation for finding a target node in circular linked list 18

4.4 Operation for deleting a node in circular linked list 18

4.5 Operation for printing nodes in circular linked list 18

5 Efficiency of an algorithm **19**
5.1 Efficiency of algorithm - Introduction to the concept . . . 19
5.1.1 Choose between two algorithms . . . 19
5.1.2 Hypothetical machine . . . 19
5.1.3 We are interested in input size . . . 20
5.2 Mathematical approach for finding the efficiency . . . 20
5.2.1 Drawbacks . . . 20
5.3 We want a theoretical way based on mathematics to compare the efficiency of the algorithms 20
5.3.1 Example of what we want as a metric . . . 21
5.3.2 Example of Big-O . . . 21
5.4 How to calculate Big-O for a given algorithm . . . 22
5.4.1 Finding Big-O . . . 22
5.5 Another approach for calculating Big-O - recurrence relationship . . . 23
5.5.1 Another example . . . 24
5.6 Another example . . . 24
5.6.1 The same but evaluated with recurrence . . . 25
5.7 Idea of best case complexity — Big Omega notation . . . 26
5.7.1 Finding Big-Ω (best case complexity) . . . 26
5.7.2 An example . . . 27
5.8 Idea of average case complexity — Big theta notation . . . 27
5.8.1 Big theta on polynomials . . . 28

6 Binary search **29**
6.1 Binary search . . . 29
6.1.1 Example . . . 29
6.1.2 Searching the upper part . . . 30
6.1.3 Searching the lower part . . . 31
6.1.4 When the target is not on the list . . . 32
6.2 Implementation . . . 32
6.3 Worst case complexity . . . 33

7 Recursion **34**
7.1 Introduction to recursion . . . 34
7.2 Basic concept of recursion . . . 34
7.3 When and how to terminate — the base condition of recursion . . . 36
7.4 Let us go into the depth of the call . . . 36
7.5 Recursion example — Juggler Sequence . . . 37
7.6 Recursion example — Finding Factorial . . . 38
7.7 Recursion example — Binary Search . . . 39
7.8 Recursion example — Decimal to Binary . . . 39
7.9 Calling a function — Operating system creates a stack . . . 40
7.10 When there is no need for a stack . . . 40
7.11 Tail recursion . . . 40
7.12 Recursion versus iteration . . . 40
7.12.1 When both are equivalent . . . 41
7.12.2 When a loop is better . . . 41
7.12.3 When recursion is better . . . 41
7.12.4 Synthesis in a programmer's way . . . 41

Chapter 1

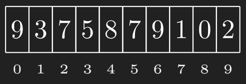
Linked List

1.1 Introduction to linked lists

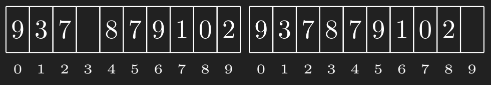
- Linked lists resolve the problems commonly encountered with arrays, that is the fact that arrays have to be declared of a fixed size and can't really be dynamically grown without fragmenting memory is a problem we can resolve using a linked list data structure.
- The draw backs of arrays are actually removed with linked lists at the cost of the benefits of having an array.

1.1.1 What is wrong with arrays?

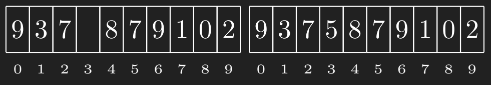
- The first issue with array is the size of the array needs to be declared. Using realloc() will not really resolve this because it will copy all the elements of the array in to a bigger memory location, this causes poor performance and fragmentation.
- Once allocated with a size it is a hazard to change the size of the array in runtime.
- Another issue is performing the insertion and deletion operation at any point between the first and the last index of the existing elements. An example can clarify this problem.



  - If we wanted to delete 5 we need to delete and shift the rest of the array to the left.



  - This takes time and impacts performance for large arrays.
  - This can happen also for insertion. Let's consider we need to insert 5 back in to the array.
  - First move everything to the right. And then insert the 5.



  - See you need to move all the $(n-1)$ elements for every insertion and deletion.
  - For this we need a linked list rather than an array.

- Adding a new element or deleting an existing element is independent to the number of elements in the linked list, no shifting kind of activities like arrays are required.

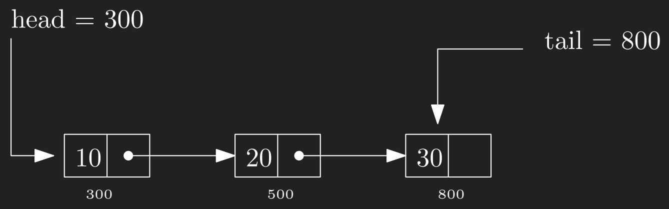
1.2 Definition of linked list, conception of node, understanding basic principles

- Linked lists are dynamic, created on "as and when required basis".
- We don't need to specify the size in a linked list.
- Elements in a linked list are discrete, not contiguous like arrays.

1.2.1 Example for storing a set of integers

- Lets declared a structure or node in this case containing members int and pointer. The int is to store the data and the pointer is to store the address of the next node.

Structure of a linked list



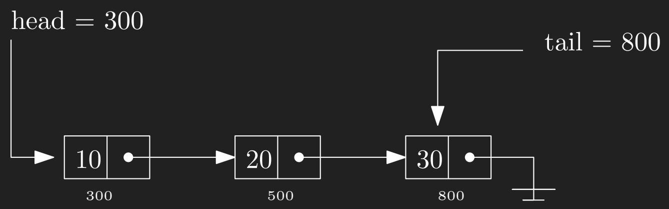
Every node stores pointers to the next or sometimes to the previous nodes.

1.3 Categories of linked lists - singly, doubly and circular linked list

1.3.1 Singly linked lists

- Consists of nodes containing a data and pointer member, each pointer points to the next node and the tail is recognized when a null pointer is encountered.

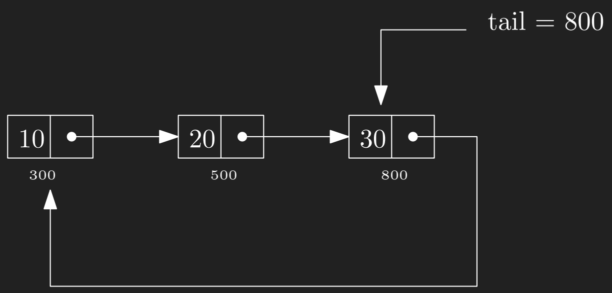
Single linked list



1.3.2 Circular linked list

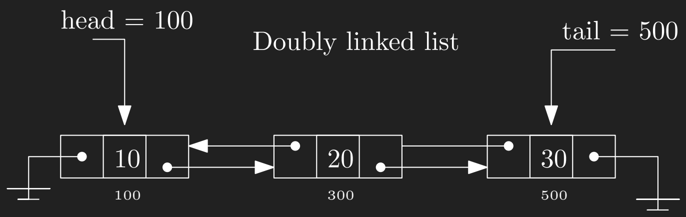
- In circular linked list there is no head, there is only a tail pointer, the tail pointer points to the last node's address.
- The terminal node does not point to null, it points to the first member of the linked list.
- This is based on the circular queue model.

Circular linked list



1.3.3 Double linked list

- In this linked list, each node has two pointers, one that points to the next node and the other that points to the previous node.
- The previous pointer in the first node and the next pointer in the terminal node are always pointing to null.
- This is a way of implementing a dynamic double ended queue.

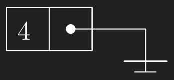


Chapter 2

Singly linked lists

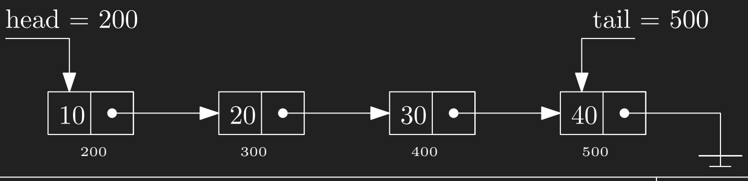
2.1 Linked list operation for insert at tail

When the linked list is empty.

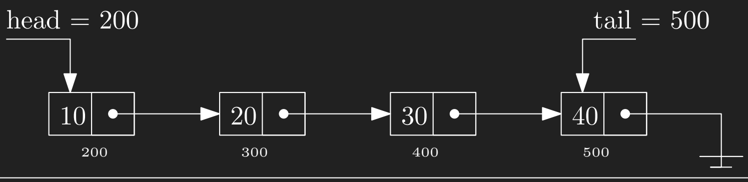


Create a new node, initialize if with data and the next pointer to null.

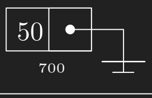
When the linked list is not empty.



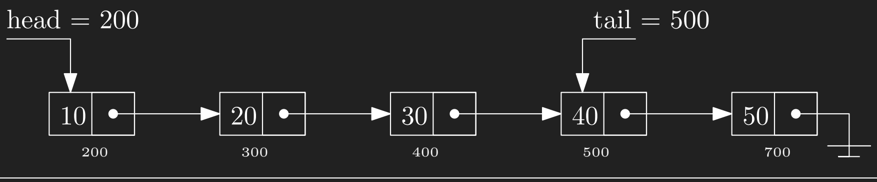
Locate the tail. Then create a new node.



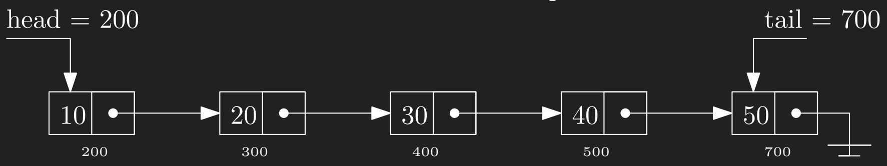
New node creation and initialization.



Modify the tail node to point to the new node.

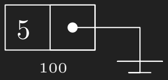


Set the tail to be the new pointer.



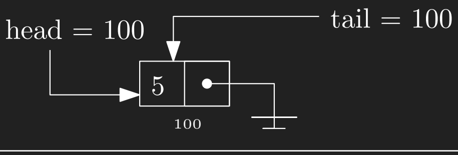
2.2 Linked list operation for inserting at head

If the linked list is empty.

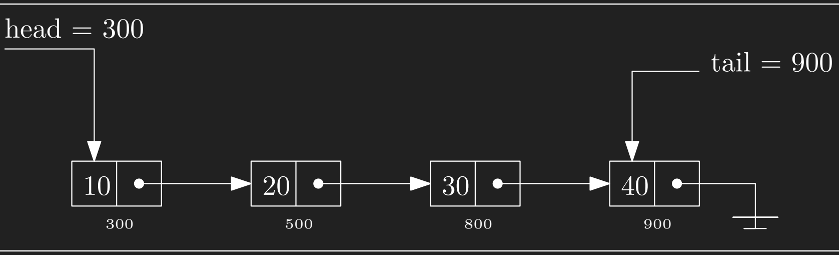


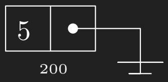
Create a new node and initialize it with data and set the next pointer to null.

Set the head and tail to point to the new node.



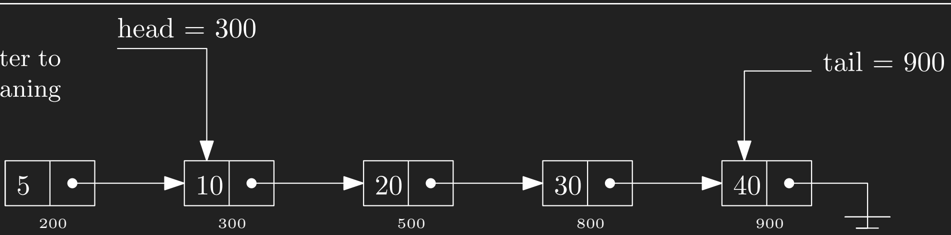
If the linked list is not empty.

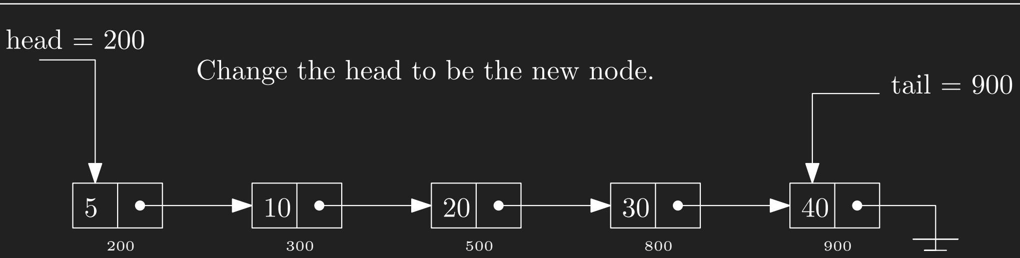




Create a new node object in memory and set the next pointer to NULL and store the data.

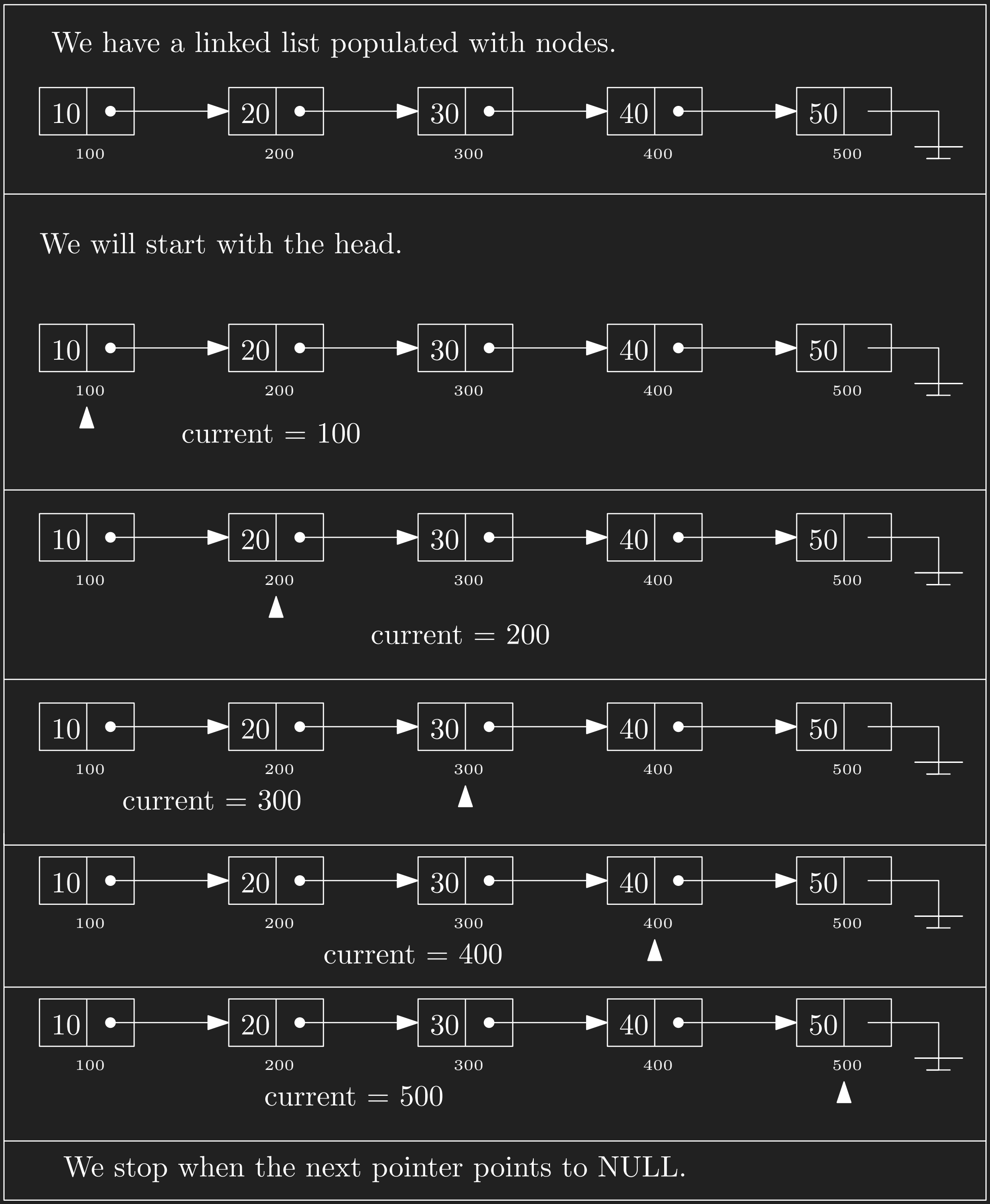
Make the new node pointer to the head of the list. Meaning newNode.next = 300.





Change the head to be the new node.

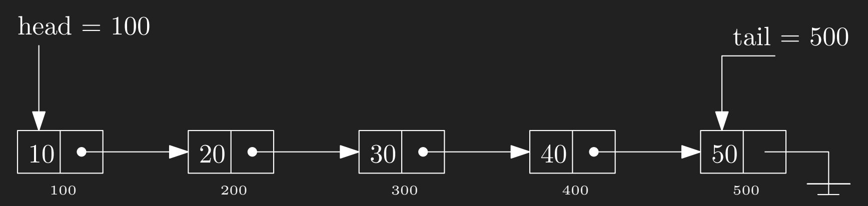
2.3 Linked list operation for traversing singly linked lists


We have a linked list populated with nodes.

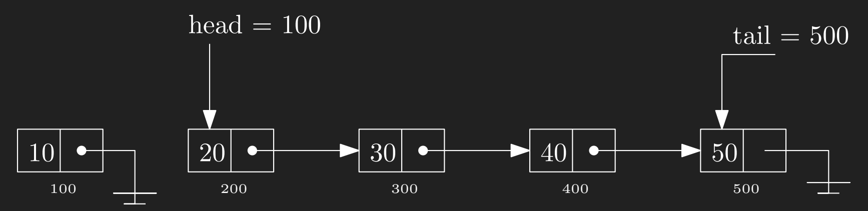
We will start with the head.

We stop when the next pointer points to NULL.

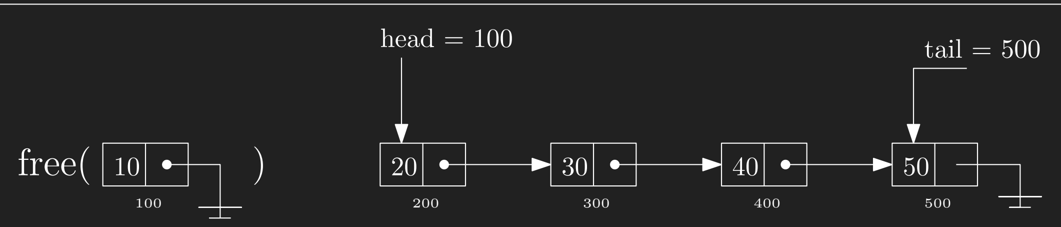
2.4 Linked list operation for delete first



Locate the head and save the pointer to the next node, in this case 200.



Set the head.next to be the head.



Free the space allocated by the first node.

2.5 Linked list operation for delete last

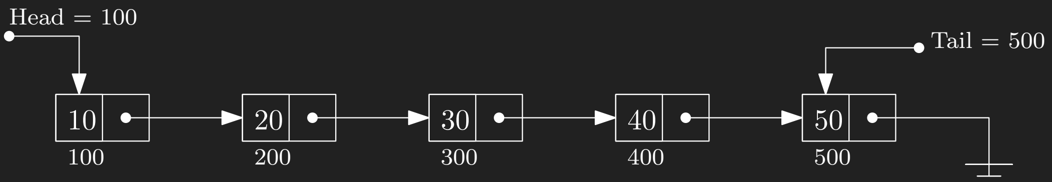
2.6 Linked list operation for deleting target nodes

2.7 Linked list operation for finding a node

2.8 Linked list operation reverse a singly linked list

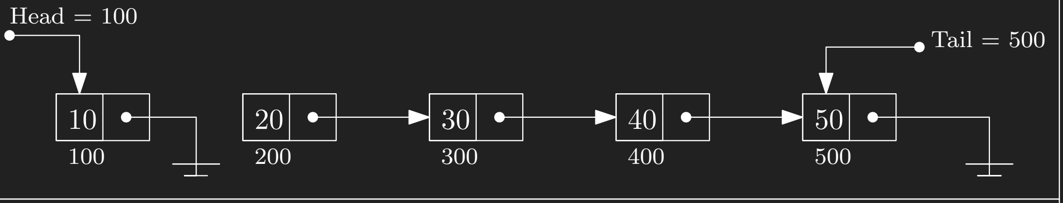
Reversing algorithm for singly linked list

Declare three variable pointers; *previous* (previous to the current), *current* will be the iterator variable and the *next* pointer.



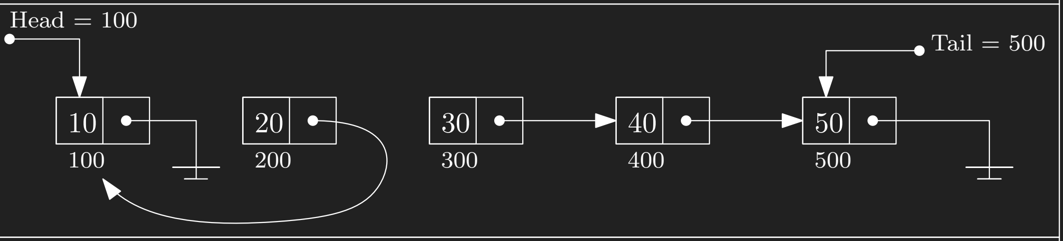
1

```
previous = NULL;
current = 100;
next = 200;
```



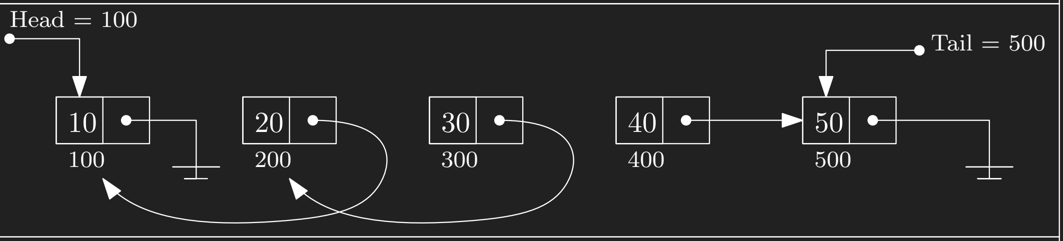
2

```
previous = 100;
current = 200;
next = 300;
```



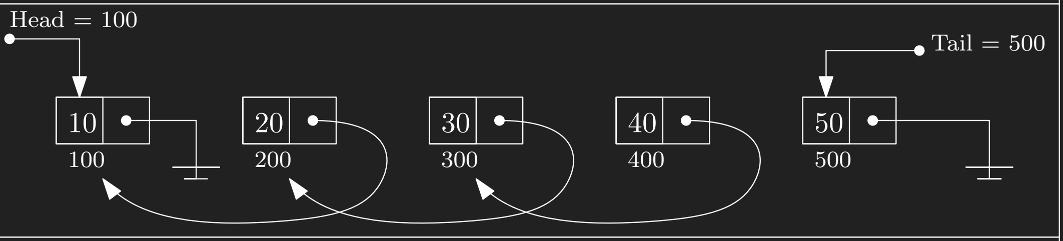
3

```
previous = 200;
current = 300;
next = 400;
```



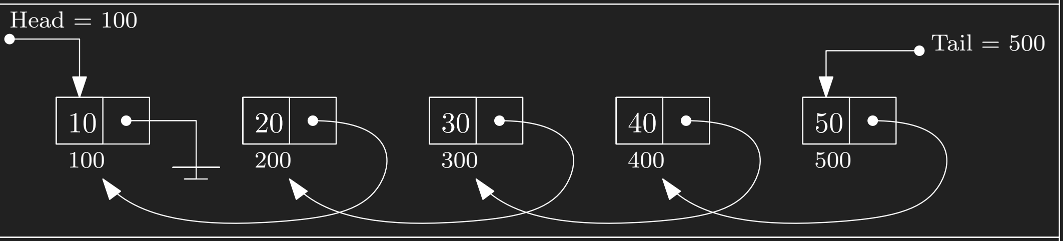
4

```
previous = 300;
current = 400;
next = 500;
```



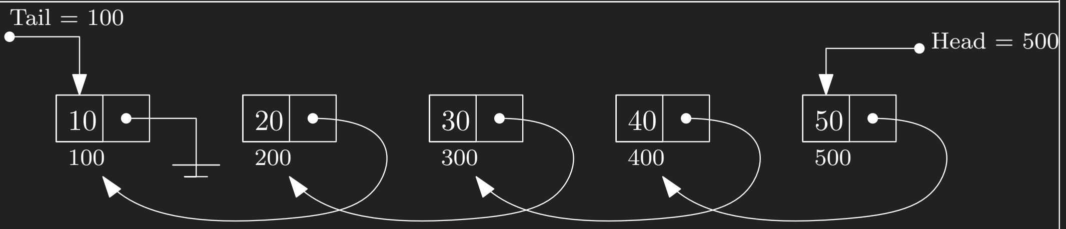
5

```
previous = 400;
current = 500;
next = NULL;
```



6

```
head = 500;
tail = 100;
```

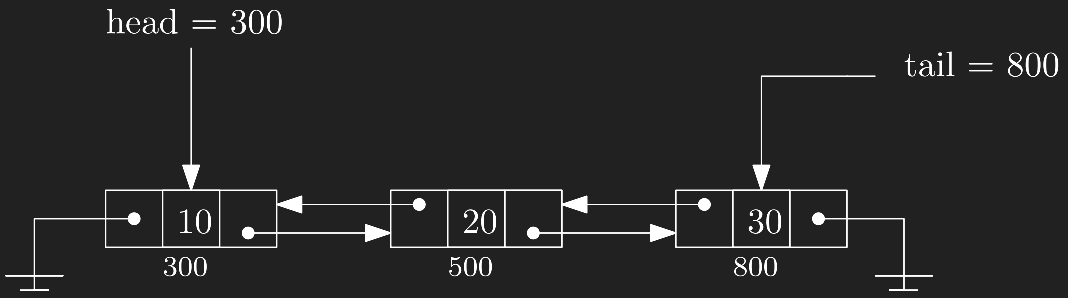


2.9 Linked list operation for printing and traversing the linked list recursively

Chapter 3

Doubly linked list

3.1 Introduction to doubly linked list



3.2 Doubly linked list operation for adding node at head

3.3 Doubly linked list operation for adding node at tail

3.4 Doubly linked list operation for finding a node in the list

3.5 Doubly linked list operation for deleting node at head

3.6 Doubly linked list operation for deleting node at tail

3.7 Doubly linked list operation for deleting a target node

3.8 Doubly linked list for traversing and printing data of the doubly linked list

Chapter 4

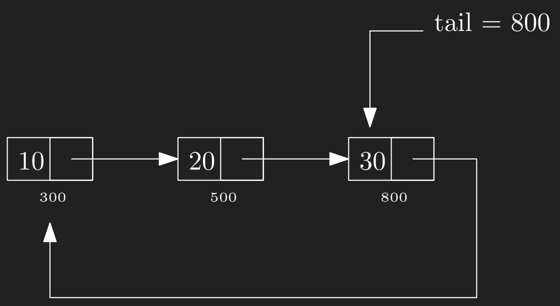
Circular linked lists

4.1 Introduction

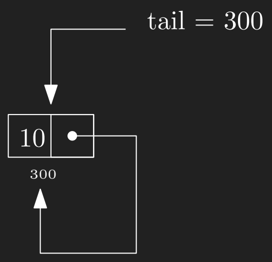
- In a circular linked lists we have pretty much the same situation as a singly linked list, the difference is that we have only one pointer to the list, the tail, and the tail.next does not point to NULL, instead it points to the first element of the list.
- We only keep track of the tail, considering the tail.next is the first node of the list.
- There is only one terminal node, no head, the tail will tell us everything we need to know.
- The best use for a circular linked list is to build a circular queue.

4.1.1 Circular linked list visualization

A populated circular linked list.



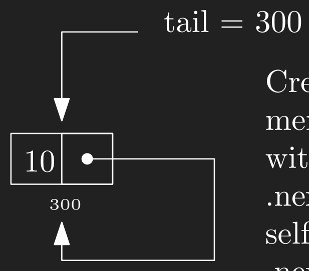
When there is only one node, the list looks like this.



- When the list is empty the first node inserted will have a .next pointing to itself.

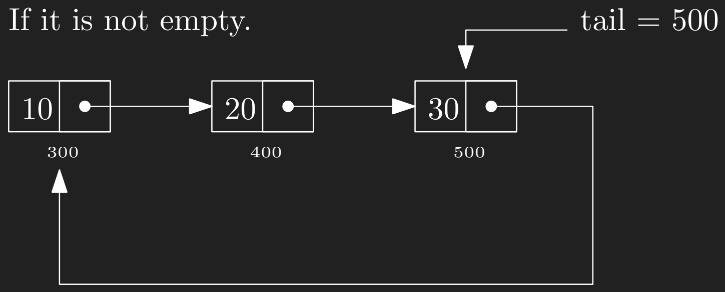
4.2 Operation for inserting a node to circular linked list

If the list is empty.

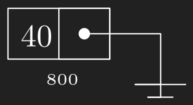


Create a new node in memory, then initialize it with data, finaly make the .next pointer point to itself, this case: data = 10, .next = 300.

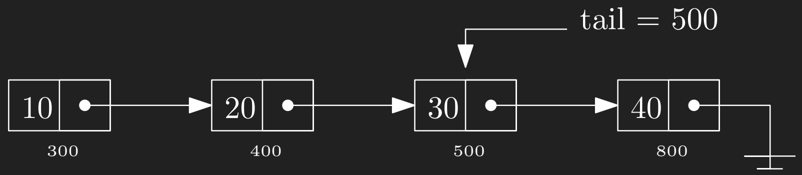
If it is not empty.



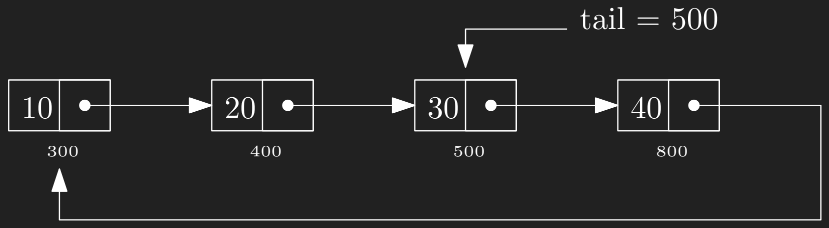
First, create a new node instance and initialize it.



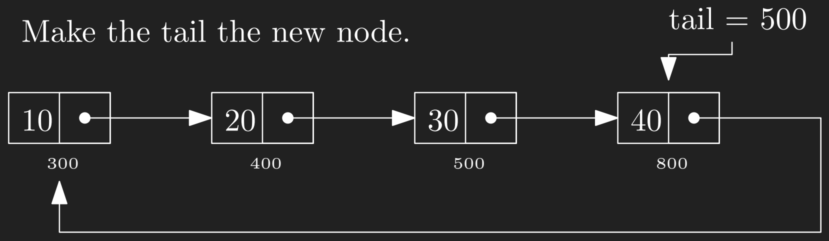
Make tail.next point to the new node.



Set the new node to point to the first member of the list.



Make the tail the new node.



4.3 Operation for finding a target node in circular linked list

4.4 Operation for deleting a node in circular linked list

4.5 Operation for printing nodes in circular linked list

Chapter 5

Efficiency of an algorithm

5.1 Efficiency of algorithm - Introduction to the concept

There are two yard sticks:

1. Execution time: this indicates the time taken by the algorithm to process data and run. It doesn't include the development time or the compilation time.

2. Memory needed by the algorithm to solve the problem: memory involved in processing the data without including the data structures themselves.

5.1.1 Choose between two algorithms

- We can compare two algorithms on the basis of execution time and the amount of memory consumed by them.

The biggest question is how to estimate the execution time and amount of memory consumed? There are basically two factors in which the exectuion time depends:

1. Machine factor: we want to keep this factor constant.

2. Input size: we'll pay much atention to this one.

5.1.2 Hypothetical machine

Let's consider a concept called RAM (Random Access Machine) model machine, this is a hypothetical machine. Where we have the following characteristics:

- No parallel processing is supported.

- Any simple instruction takes one unit of time.

- Loops and subroutines are not simple operations.

- Infinite memory.

We will just sum up the amount of instructions and deduce the time taken by the machine.

5.1.3 We are interested in input size

Given an input size the greater it is the more time the algorithm is going to take, thus we have to Given the machine factor being constant with our hypothetical RAM model.

What we want is essentially to make our algorithm be independent on input size, but this is virtually imposible, so we must seek to reduce execution time as much as possible. We can calculate the time taken by a function if $f(n)$.

$$T = f(n)$$

Where n is an integer and $n > 0$.

5.2 Mathematical approach for finding the efficiency

One way is to implement, run and find execution time, then use a time tracking algorithm to calculate time.

Consider two algorithms:

```
function f1(n) {
    <code>
}
function f2(n) {
    <code>
}
```

For testing the time taken by the algorithms we can do this:

```
t1 = time();
f1(5000);
t2 = time();
T = t2 - t1;


t1 = time();
f2(5000);
t2 = time();
T = t2 - t1;
```

We can now check which algorithm is better.

5.2.1 Drawbacks

1. This method however will not allow us to draw a general conclusion about execution time depending on n, we are not finding execution time for all possible input size n but only for some input size. It is a fact that n is a positive integer and can be as large as possible.

2. This approach requires an implementation, and this can be expensive because it requires man-hours.

5.3 We want a theoretical way based on mathematics to compare the efficiency of the algorithms

- We need some sort of way of determining a limit such as a function will never exceed the limit $T \leq c \times n^2$

5.3.1 Example of what we want as a metric

$T_1 \leq c \times n^2$

```
function f1(n){
    <code>;
}
```

$T_2 \leq c \times n \log(n)$

```
function f2(n){
    <code>;
}
```

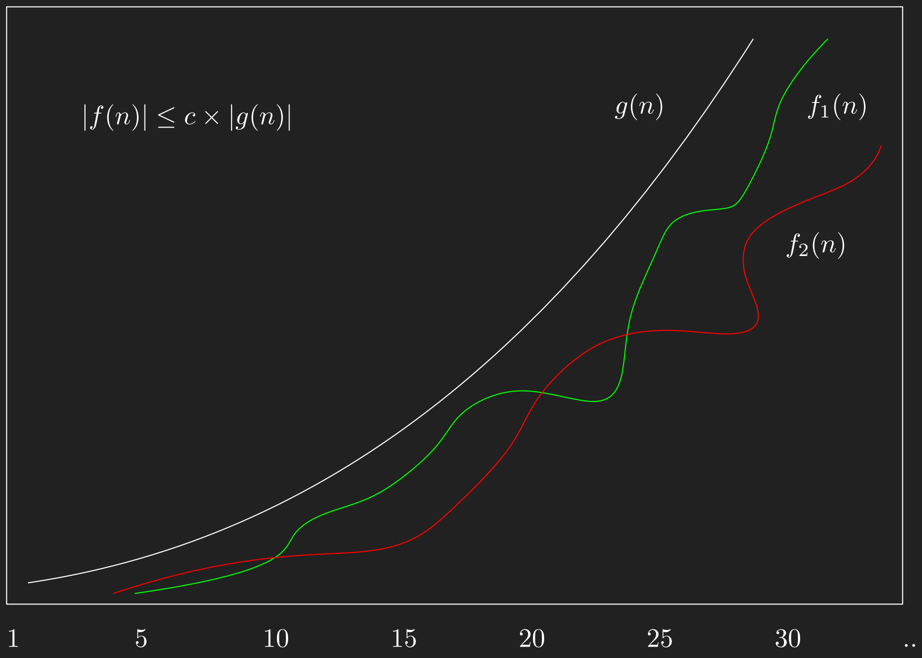
- Defining c as a positive integer constant.
- For any algorithm, if you find that the complexity is something like $f(n) \leq c \times n^2$ we say that the complexity $O(n^2)$ (Big-O of n^2) this is the worse case scenario.
- For any algorithm we find that has a complexity of $f(n) \leq c \times n \log(n)$ we say that the algorithm has a worse case scenario of $O(n \log(n))$ (Big-O of $n \log(n)$).
- If an algorithm has a complexity of $f(n) \leq c \times 2^n$ (Big-O of 2^n) this is the worse case scenario.
- If however the algorithm is said to have $O(\log(n))$ we understand that the algorithm will never exceed $f(n) \leq c \times \log(n)$ complexity.
- Generalizing the concept:

$$O(g(n)) \implies f(n) \leq c \times g(n)$$

  - Meaning that the algorithm will never take more execution time than $c \times g(n)$, $g(n)$ is any function determined to be the complexity of the algorithm.

5.3.2 Example of Big-O

A function $f(n)$ is said to be $O(g(n))$ if and only if there exists a positive constance c and non negative integer n_0 such that $|f(n)| \leq c \times |f(n)|$, it is said that $f(n) \in O(f(n))$



5.4 How to calculate Big-O for a given algorithm

Before anything lets understand the general polynomial.

$$a_k(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

For any general polynomial we can prove that the value of the polynomial is:

$$\left|a_(n)\right| \leq c \times n^k, \quad n > n_0$$

For example, we have the following polynomial:

$$f(n) = 2n^2 + 3n + 5 \quad \text{The highest order of } n \text{ is 2.}$$
$$|f(n)| \leq c \times n^2$$

Another example:

$$f(n) = 4n^3 + 2n^2 + 3n + 7$$
$$|f(n)| \leq c \times n^3$$

If you are given something like this:

$$f(n) = n \log(n) + n + 5$$
$$|f(n)| \leq c \times n \log(n)$$

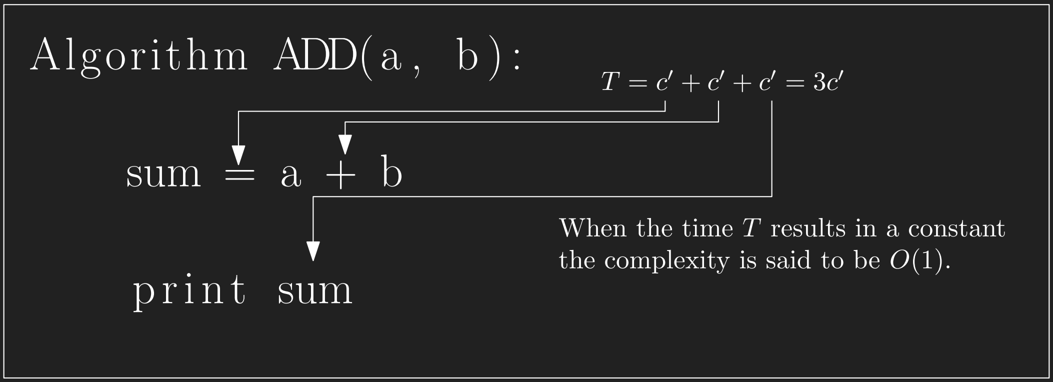
Another example:

$$f(n) = 4n + 5$$
$$|f(n)| \leq 4n + n \quad , n > 5$$

5.4.1 Finding Big-O

We will execute the algorithm using the RAM model hypothetical machine, where each instruction takes a constant amount of time.

When finding Big-O you need to account for every instruction, assignments take one unit of time, represented by c', arithmetic operators take one unit of time as well.

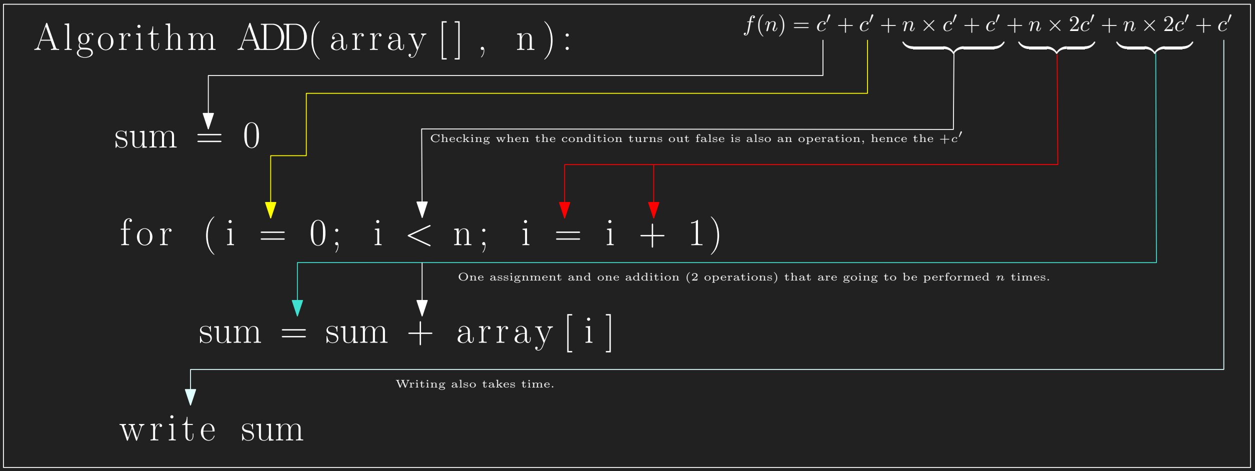


```
Algorithm ADD(a, b):

    sum = a + b

    print sum
```

$T = c' + c' + c' = 3c'$

When the time T results in a constant the complexity is said to be $O(1)$.

The previous calculation ended in a constant complexity, this means that it doesn't matter the size of the input it will always take the same or constant amount of time to preform the operation. The next one, as we can see does not result in constant complexity, it is indeed dependent on the input.



$$f(n) = c' + c' + \underbrace{n \times c' + c'} + \underbrace{n \times 2c'} + \underbrace{n \times 2c'} + c'$$

```
Algorithm ADD(array[], n):
        sum = 0
        for (i = 0; i < n; i = i + 1)
            sum = sum + array[i]
        write sum
```

Checking when the condition turns out false is also an operation, hence the $+c'$

One assignment and one addition (2 operations) that are going to be performed n times.

Writing also takes time.

$$f(n) = c' + c' + n \times c' + c' + n \times 2c' + n \times 2c' + c' = 4c' + n \times 5c'$$

Since the highest order of n is 1, this means the complexity is $O(n)$.

As you can see the biggest contributor for the execution time is the for loop.

The time taken for is some constant times n thus we can conclude $f(n) \leq O(n)$ which means the complexity is $O(n)$.

5.5 Another approach for calculating Big-O - recurrence relationship

Suppose the same algorithm above, we can find the recurrent relationship which is to say do the following:

$$\begin{aligned} f(n) &= c + f(n-1) \\ f(n-1) &= c + f(n-2) \end{aligned}$$

- Substitute $f(n-1)$ into $f(n)$.

$$\begin{aligned} f(n) &= c + (c + f(n-2)) \\ f(n) &= 2c + f(n-2) \\ f(n-2) &= c + f(n-3) \end{aligned}$$

- Do the same and substitute $f(n-2)$ into $f(n)$.

$$
\begin{aligned}
f(n) &= 2c + (c + f(n-3)) \\
f(n) &= 3c + f(n-3) \\
&\vdots \\
f(n) &= (n-1) * c + f(n-(n-1)) \\
f(n) &= (n-1) * c + f(1) \\
\therefore f(n) &= (n-1) * c + c
\end{aligned}
$$

5.5.1 Another example

```
void print(int n){
    if (n <= 0){
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

5.6 Another example

```
Algorithm Process (n):
    for (i = n; i >= 1; i = i / 2):
        < Some simple operations>
```

As you can see, there is a noticeable pattern.

$$
\begin{aligned}
i &= n = \frac{n}{2^0} \\
i &= \frac{n}{2^1} \\
i &= \frac{n}{2^2} \\
&\vdots \\
i &= \frac{n}{n} = \left(\frac{n}{2^k}\right)
\end{aligned}
$$

- We can find k or the number of times this for loop will repeat.

$$\begin{aligned}\frac{n}{2^k} &= 1\\ n &= 2^k\\ k &= \log_2(n)\end{aligned}$$

Thus the total time taken for this algorithm is:

$$\begin{aligned}T &= c \times \log_2(n) + c\\ T &= c' \log_2(n)\end{aligned}$$

Thus we can say that the complexity of this algorithm is $O(\log(n))$.

5.6.1 The same but evaluated with recurrence

Suppose the same algorithm presented above (Process (n)). Let's define the function.

$$\begin{aligned}f(n) &= c + f\left(\frac{n}{2}\right)\\ f\left(\frac{n}{2}\right) &= c + f\left(\frac{n}{2^2}\right)\end{aligned}$$

- Substitute $f(n/2)$ into $f(n)$.

$$\begin{aligned}f(n) &= c + c + c\left(\frac{n}{2^2}\right)\\ &= 2c + \left(\frac{n}{2^2}\right)\\ &\vdots\\ &= kc + \left(\frac{n}{2^k}\right)\\ &= \log_2(n) \times c + f\left(\frac{n}{2^{\log_2(n)}}\right)\\ &= \log_2(n) \times c + f\left(\frac{n}{n}\right)\\ &= \log_2(n) \times c + c\end{aligned}$$

- It will iterate as long as it is not 1 or $f(1)$.
- We can say now that the complexity of this algorithm is:

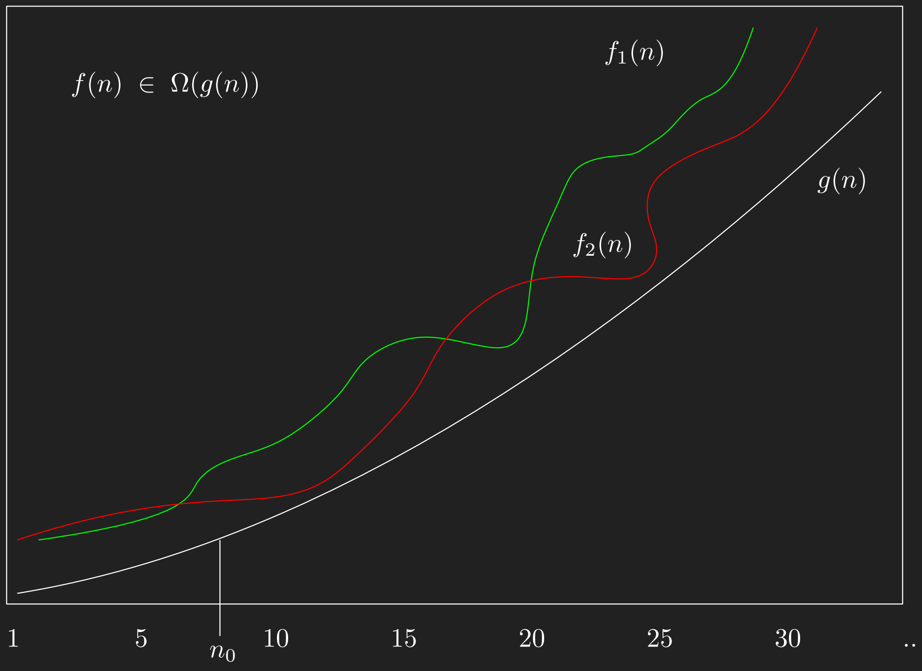
$$\begin{aligned}f(n) &\leq c' \times \log_2(n)\\ &\therefore O(\log(n))\end{aligned}$$

5.7 Idea of best case complexity — Big Omega notation

- Unlike the Big-O analysis, the best case analyzes the minimum amount of time that the algorithm will always take.
- It is important to consider that that minimum time will occur in special cases or depending on the input size.
- Def: A function $f(n)$ is said to be $\Omega(g(n))$ if and only if there exists a positive constant C and a non-negative integer n_0 such that:

$$|f(n)| \geq C \times g(n), \; \forall \; n \geq n_0$$

- This minimum time will happen only for special scenarios.



5.7.1 Finding Big-Ω (best case complexity)

In the following example, searching for 56 is going to require traversing the whole array and returning -1. The worst case scenario or Big-O, the worst case happens when the item is not found for this algorithm the worst case complexity is $O(n)$.

```
A[] = [10,20,30,40,50]

Algorithm linear_search (A[], n, target):
    for (i = 0; i < n; i = i + 1):
        if (A[i] == target):
            return i
    return -1
```

However, to find 10, in the array, it happens to be the first index, thus we can say that the algorithm's best case complexity is $\Omega(1)$, this is because it takes constant time to find 10, and we assume the best scenario.

5.7.2 An example

On the following we must find the best case scenario. For this we evaluate the case in which we can take as little amount as possible.

```
Algorithm Process (n):
    f = 1
    for (i = 1; i <= n && f == 1; ++1):
        input k
        sum = 0
        for (j = 1; j <= n; j++):
            sum = sum + k
            if (k is odd):
                f = 0
            end if
        end for
        write sum
    end for
```

The worse case complexity is an order of n^2 or $O(n^2)$, this is because depending on the variable k the amount of times to iterate is determined, k might never be odd, and this gives us an order of n^2.

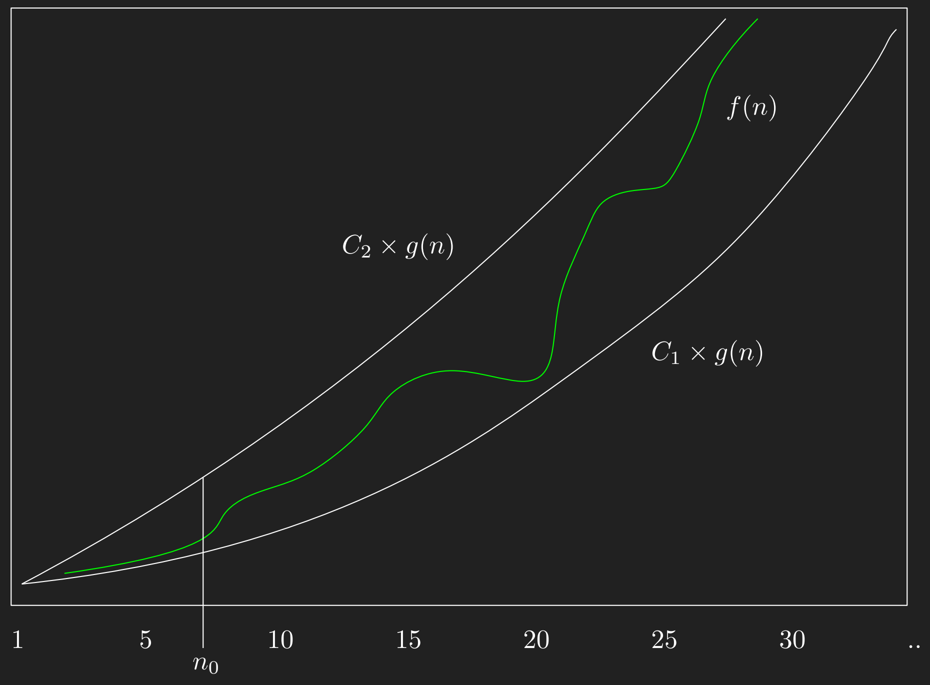
On the other case, if in the very first iteration k is inputted to be odd, this will break the outer loop condition and make it so that only the inner loop executes n times, this gives us the best case complexity of $\Omega(n)$.

5.8 Idea of average case complexity — Big theta notation

- Def. A function $f(n)$ is said to be $\Theta(g(n))$ if and only if there exists two positive constants C_1 and C_2 and non-negative integer n_0 such that:

$$C_1 \times g(n) \leq |f(n)| \leq C_2, \forall\, n \geq n_0$$

- Big-Theta is considered to be more accurate than the others (Big-O and Big-Omega).



5.8.1 Big theta on polynomials

$$f(n) = \frac{1}{2}n^2 - 3n \quad f(n) \in \Theta(n^2)$$
$$C_1 \times n^2 \leq \frac{1}{2}n^2 - 3n \leq C_2 \times n^2$$

- Find the inequalities.

$$C_1 \times n^2 \leq \frac{1}{2}n^2 - 3n, \quad n \geq 12$$
$$C_1 \leq \frac{1}{4}, \quad C_1 = \frac{1}{8}$$
$$\frac{1}{8}n^2 \leq \frac{1}{2}n^2 - 3n$$

$$\frac{1}{2}n^2 - 3n \leq C_2 \times n^2$$
$$\frac{1}{2} - \frac{3}{n} \leq C_2 \qquad n \geq 12$$
$$\frac{1}{2} - \frac{3}{12} \leq C_2$$
$$\frac{1}{2} - \frac{1}{4} \leq C_2$$
$$C_2 = 1 \quad C_2 = 2$$
$$\frac{1}{8}n^2 \leq \frac{1}{2}n^2 - 3n \leq n^2$$
$$f(n) \in \Theta(n^2) \ \forall \ n \geq 12$$

Chapter 6

Binary search

6.1 Binary search

- Before, we have seen the linear search algorithm which traverses an array, of which worse case complexity was $O(n)$.

- In case of binary search, the worse case complexity is narrowed down to $O(\log(n))$.

- Now this algorithm works for looking for target data in a list or array, the array has to be in sorted order, ascending or decending order, but in order.

- Binary search is best applied when the elements are kept in a binary search tree.

- For a binary search tree no overhead is here for keeping the elements in sorted order.

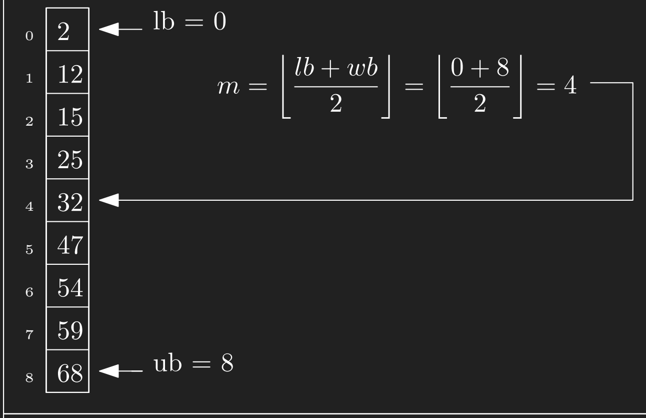
- The list or array must be in order, ascending or descending, but in order.

6.1.1 Example

- Consider variables lower bound and upper bound, these hold the current bounds in which the array will be searched. Suppose we want to find 59 in the list.

6.1.2 Searching the upper part

Search for target 59 in the following list of size 9.



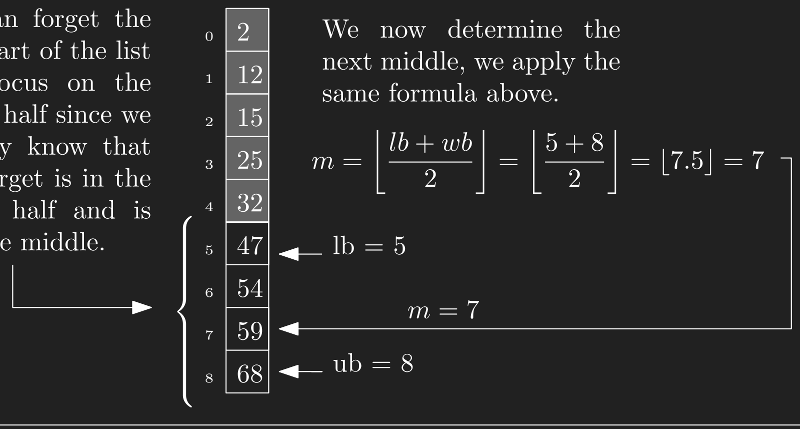
$$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{0+8}{2} \right\rfloor = 4$$

Compare the middle element and check whether it contains the target (59). If it doesn't contain the target compare if the target is bigger or smaller than the middle element; if the target is greater the middle might be in the upper half, if the target is less than the middle then the target might be in the lower half.

```
if $middle.data == target$ then
    we have found the target;
else if $middle.data < target$ then
    the target might be in the lower half;
else if $middle.data > target$ then
    the target might be in the upper half;
end
```

Since we have determined that list[4] is 32 and the target 59, the target is clearly larger than the middle, thus we understand that the target can only be in the upper half of the list, thus we only have to search the upper half.

We can forget the first part of the list and focus on the upper half since we already know that the target is in the upper half and is not the middle.



We now determine the next middle, we apply the same formula above.

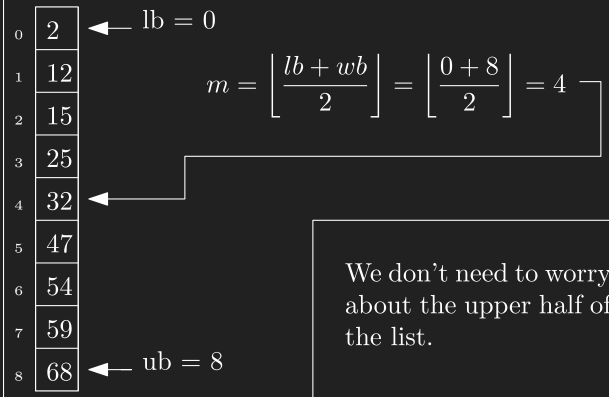
$$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{5+8}{2} \right\rfloor = \lfloor 7.5 \rfloor = 7$$

Now we compare if the middle is the target, it indeed is, we now have found the target in the list with as little operations as posible.

Target is contained in the 7^{th} index of the list.

6.1.3 Searching the lower part

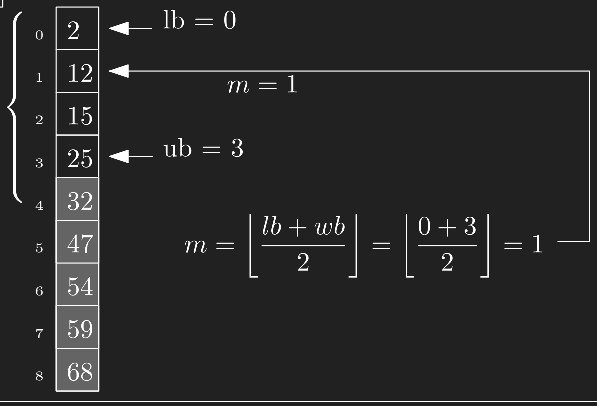
Now search for target 15 in list of size 9.



0 2
1 12
2 15
3 25
4 32
5 47
6 54
7 59
8 68
lb = 0
ub = 8
$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{0 + 8}{2} \right\rfloor = 4$

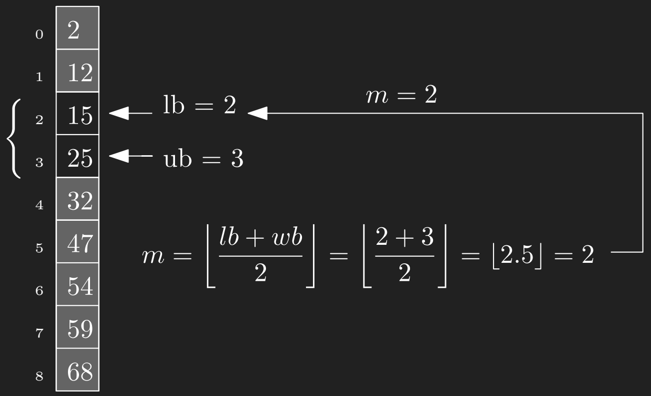
Again we determine the middle to be 4 which contains 32, we can see that 32 is not 15, and 15 is less than 32, thus we know that the target must be in in the lower half. We move the upper bound to middle minus one index.

We don't need to worry about the upper half of the list.



0 2
1 12
2 15
3 25
4 32
5 47
6 54
7 59
8 68
lb = 0
$m = 1$
ub = 3
$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{0 + 3}{2} \right\rfloor = 1$

We again compare the new middle with the target, since the middle is 12 and the target is 15 the target is clearly grater, thus we move the lower bound up to the middle + 1 and we define a new middle.



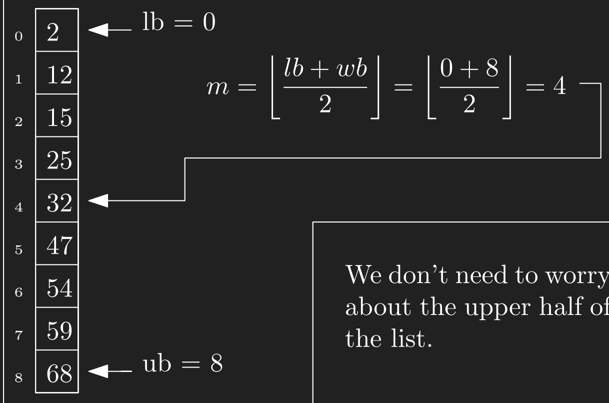
0 2
1 12
2 15
3 25
4 32
5 47
6 54
7 59
8 68
lb = 2
$m = 2$
ub = 3
$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{2 + 3}{2} \right\rfloor = \lfloor 2.5 \rfloor = 2$

The middle is list[2] which is 15, the target is 15, we have found the target uppon comparing the middle to the target.

Therefore, the target is contained at index 2.

6.1.4 When the target is not on the list

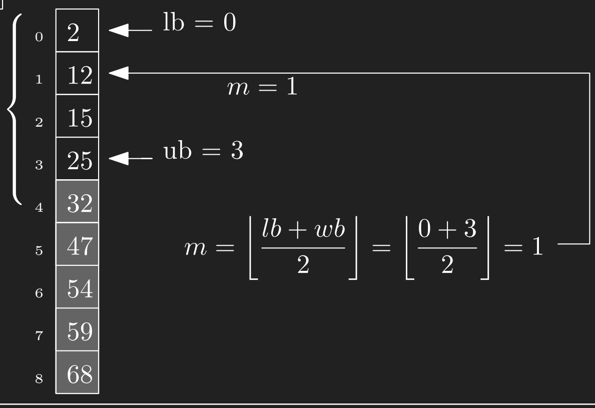
Now search for target 15 in list of size 9.



$$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{0+8}{2} \right\rfloor = 4$$

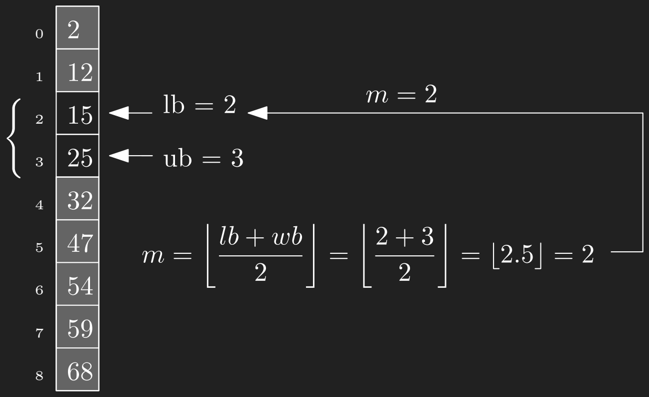
Again we determine the middle to be 4 which contains 32, we can see that 32 is not 15, and 15 is less than 32, thus we know that the target must be in in the lower half. We move the upper bound to middle minus one index.

We don't need to worry about the upper half of the list.



$$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{0+3}{2} \right\rfloor = 1$$

We again compare the new middle with the target, since the middle is 12 and the target is 15 the target is clearly grater, thus we move the lower bound up to the middle + 1 and we define a new middle.



$$m = \left\lfloor \frac{lb + wb}{2} \right\rfloor = \left\lfloor \frac{2+3}{2} \right\rfloor = \lfloor 2.5 \rfloor = 2$$

The middle is list[2] which is 15, the target is 15, we have found the target uppon comparing the middle to the target.

Therefore, the target is contained at index 2.

6.2 Implementation

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

// function prototypes
int binary_search(int[] , int, int);

// implementation
int binary_search(int list[] , int size, int target){
    int lower_bound = 0, upper_bound = size - 1;
    while (lower_bound <= upper_bound){
        int middle = (int)(lower_bound + upper_bound)/2;
        if (target == list[middle]){
            return middle;
        } else if (target > list[middle]){
            lower_bound = middle + 1;
        } else {
            upper_bound = middle - 1;
        }
    }
    return -1;
}
int main() {
    int list[] = {45,89,267,984,1256,1489};
    int b = binary_search(list,sizeof(list)/sizeof(int),1256);
    b != -1 ? printf("Found at %d => %d\n",b,list[b]) : printf("Not found.\n");
    return 0;
}
/* Output:
Found at 4 => 1256
*/
```

6.3 Worst case complexity

On each iteration the n is split in two, $n \to n/2 \to n/2^2$ and eventually we'll get to $n/2^i = n/n = 1$ this will happen when $i = \log_2(n)$ there is only one comparison happening in each iteration because of the if,else if and else; since this is the case there are $\log(n)$ comparisons. Thus, the Big-O is $O(\log(n))$.

Chapter 7

Recursion

7.1 Introduction to recursion

- Anything done with recursion can be done with a loop.
- Memory sided way in which recursion goes on.
- Compare between recursive and iterative approaches of problem solving.
- Tail recursion.

It is recommended to try to understand recursion in an abstract way.

7.2 Basic concept of recursion

A function that calls itself until a base condition is met.

- The following program is an example of a recursive function call, the problem is that the function will call itself infinitely, we don't have anything to actually break out of the recursive call.

```c
void print(){
    printf("Hello!!\n");
    print(); // recursive call
}
```

- Suppose another example, we want to print a variable k inside the print function, can you guess what will it print as the value of k?

```c
void print(){
    int k = 1;
    printf("Hello!!,k = %d\n",k);
    k++;
    print(); // recursive call
}
/* Output:
Hello!!,k = 1
Hello!!,k = 1
Hello!!,k = 1
Hello!!,k = 1
...
*/
```

- Even though we increment k before calling the function again recursively we still get 1, this is because upon calling the function again we are creating. k is allocated every time the function calls itself.

- We can solve this by making the k variable static, this means that you don't want to allocate a new space for k every time the function calls itself but rather the space allocated will be always the same.
    - Static variables are meant to reference only one allocated space during the entirety of the program and exist once during the entirety of the program.

```c
void print(){
    static int k = 1;
    printf("Hello!!,k = %d\n",k);
    k++;
    print(); // recursive call
}
/* Output:
Hello!!,k = 1
Hello!!,k = 2
Hello!!,k = 3
Hello!!,k = 4
...
*/
```

- As we can see k is incrementing now because the k variable has been declared static.

- The recursive function above will never terminate, it doesn't have a case in which you want to break the recursion, we can use this with a simple if-else statement, and to this if-else statement we give the name as the *base case* which will break the loop.

```c
void print(){
    static int k = 1;
    printf("Hello!!,k = %d\n",k);
    k++;
    if (k <= 3){
        print(); // recursive call
    } else {
        return;
    }
}
/* Output:
Hello!!,k = 1
Hello!!,k = 2
Hello!!,k = 3

*/
```

- Remember to look at this abstractly, if you inspect all the function calls at a low level you will find it more difficult to understand recursion.

- Usually we don't use recursion like this, k would be a parameter and the function would be called with variable parameters.

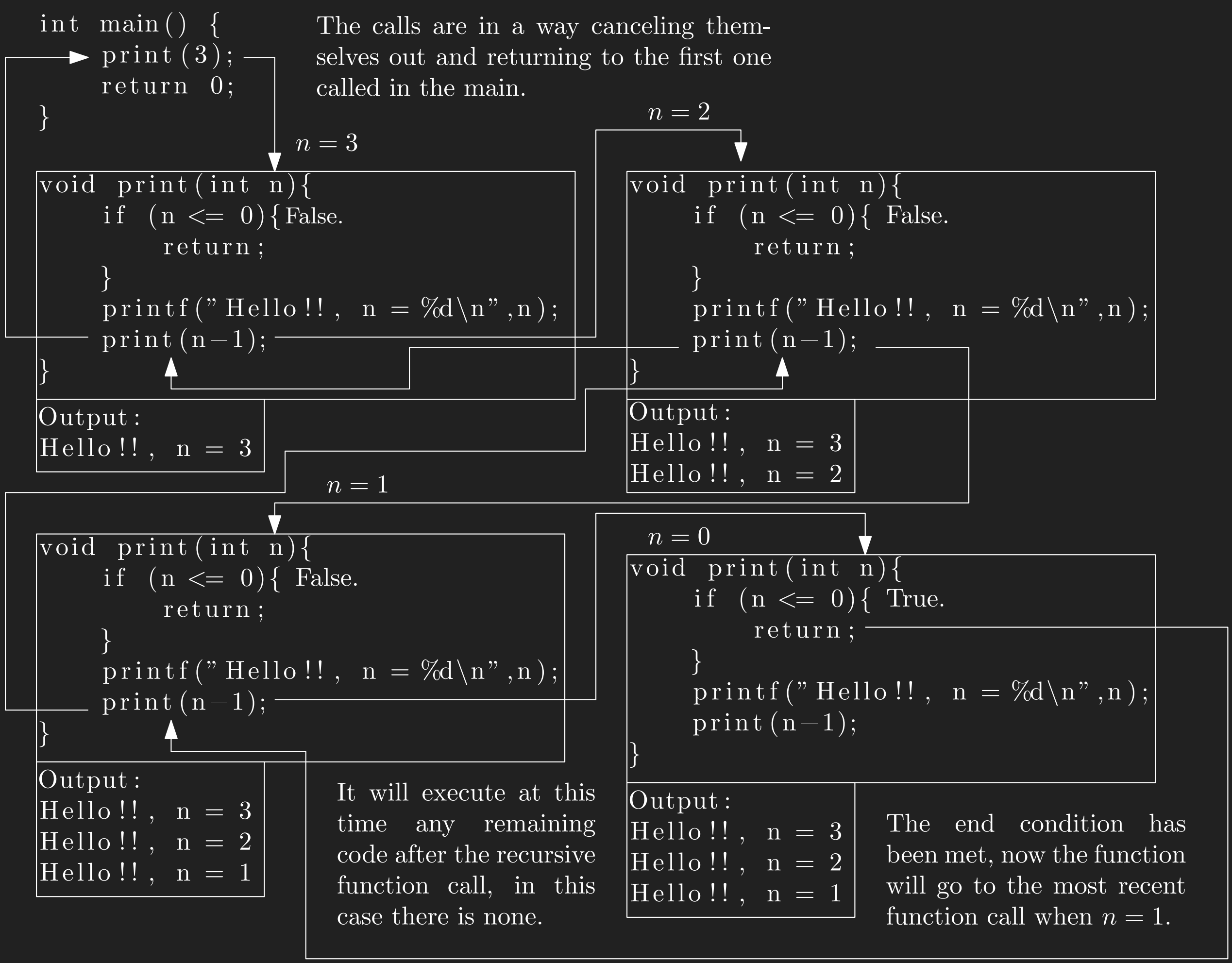
7.3 When and how to terminate — the base condition of recursion

- If we were to print a variable number of times we must accept a parameter and based on that execute the function.

```c
void print(int n){
    if (n <= 0){
        return;
    }
    printf("Hello!!\n");
    print(n-1); // recursive call
}
```

  - In order to meet the base condition and eventually terminate the recursive calls

7.4 Let us go into the depth of the call



```
int main() {
    print(3);
    return 0;
}
```

The calls are in a way canceling themselves out and returning to the first one called in the main.

$n = 3$

```
void print(int n){
    if (n <= 0){ False.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
```

$n = 2$

```
void print(int n){
    if (n <= 0){ False.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
Hello!!, n = 2
```

$n = 1$

```
void print(int n){
    if (n <= 0){ False.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1
```

It will execute at this time any remaining code after the recursive function call, in this case there is none.

$n = 0$

```
void print(int n){
    if (n <= 0){ True.
        return;
    }
    printf("Hello!!, n = %d\n",n);
    print(n-1);
}
```

```
Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1
```

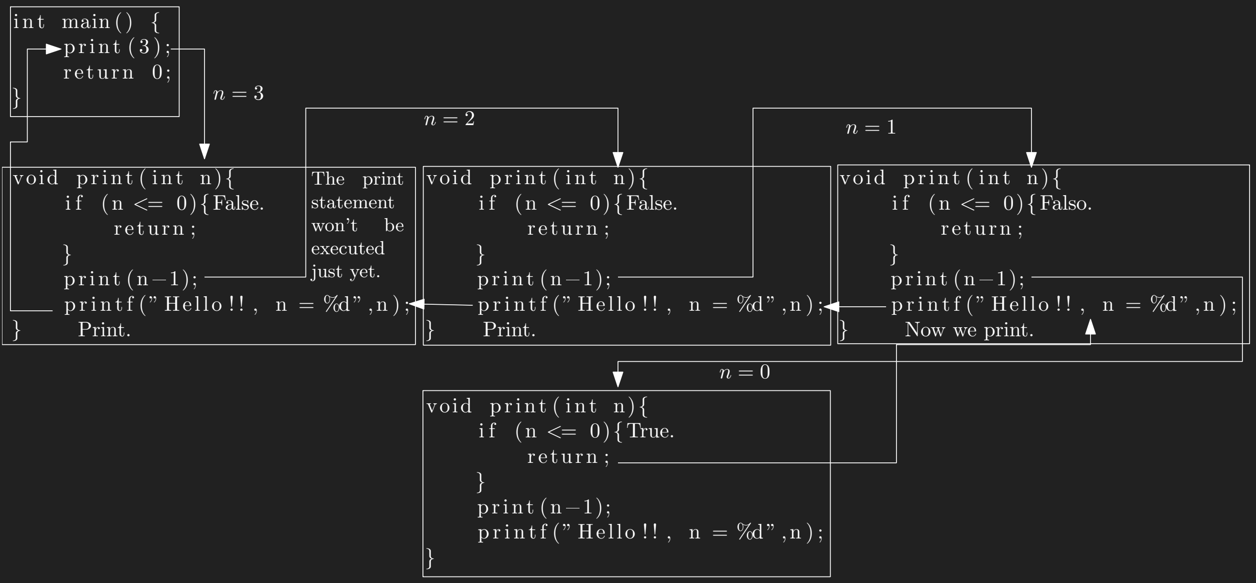
The end condition has been met, now the function will go to the most recent function call when $n = 1$.

- If you were to place anything below the recursive call it will be executed after the calls have been made.

```c
    void print(int n){
    if (n <= 0){
        return;
    }
    printf("Hello!!\n");
    print(n-1); // recursive call
}
/* Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1

*/
```

- If we were to make the print statement after the recursive call the order will be inverted.

```c
void print(int n){
    if (n <= 0){
        return;
    }
    print(n-1); // recursive call
    printf("Hello!!\n");
}
/* Output:
Hello!!, n = 1
Hello!!, n = 2
Hello!!, n = 3

*/
```



```c
int main() {
    print(3);
    return 0;
}
```

```c
void print(int n){
    if (n <= 0){
        return;
    }
    print(n-1);
    printf("Hello!!, n = %d",n);
}
```

7.5 Recursion example — Juggler Sequence

https://en.wikipedia.org/wiki/Juggler_sequence

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

```c
#include <math.h>

//  asume the congecture 10^6 nums end at one.
void juggler(int a){
    // base condition.
    if (a == 1){
        printf("%d\n",a);
        return;
    }
    printf("%d, ",a);
    juggler((a % 2) != 0 ? (int)pow(a,1.5) : (int)pow(a,0.5));
}
void juggler_rev(int a){
    // base condition.
    if (a == 1){
        printf("%d, ",a);
        return;
    }
    juggler_rev((a % 2) != 0 ? (int)pow(a,1.5) : (int)pow(a,0.5));
    printf("%d, ",a);
}
int main() {
    printf("Juggler(3): ");
    juggler(3);
    printf("Juggler(3) reverse: ");
    juggler_rev(3);
    return 0;
}
/* Output:
Juggler(3): 3, 5, 11, 36, 6, 2, 1
Juggler(3) reverse: 1, 2, 6, 36, 11, 5, 3,
*/
```

7.6 Recursion example — Finding Factorial

https://es.wikipedia.org/wiki/Factorial#:~:text=Podemos%20definir%20el%20factorial%20de,menores%20o%20iguales%20que%20n.

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

unsigned int factorial(unsigned int n){
    if (n == 0){
        return 1;
    }
    return n * factorial(n-1);
}

int main() {
    printf("%u",factorial(5));
    return 0;
}
```

7.7 Recursion example — Binary Search

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

int bin_search(int arr[], int lb, int ub, int target){
    int middle = (int)((lb + ub) / 2);
    if (lb > ub) {
        return -1;
    }
    int m = (int)((lb + ub) / 2);
    if (arr[m] == target){
        return m;
    } else if (arr[m] > target){
        bin_search(arr,lb,m-1,target);
    } else {
        bin_search(arr,m+1,ub,target);
    }
}

int main() {
    int arr[] = {10,20,30,40,50,60,70,80,90};
    const int target = 80;
    int k = bin_search(arr,0,sizeof(arr)/sizeof(int)-1,target);
    k != -1? printf("Found at %d => %d",k,target): printf("Not found.");
    return 0;
}
/* Output:
Found at 7 => 80
*/
```

7.8 Recursion example — Decimal to Binary

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

void decToBin(unsigned n){
    if (n == 0) {
        printf("0");
        return;
    } else if (n == 1){
        printf("1");
        return;
    }
    int r = (int)(n % 2);
    n = n / 2;
    decToBin(n);
    printf("%d",r);
}
int main() {
    decToBin(67);
    printf("\n");
    decToBin(90);
    printf("\n");
    decToBin(1);
    printf("\n");
    decToBin(0);
    printf("\n");
    decToBin(5);
    printf("\n");
    return 0;
}
/* Output:
1000011
1011010
1
0
101

*/
```

7.9 Calling a function — Operating system creates a stack

- The operating system creates a stack and pushes all the function calls. Then as the functions are executed they get popped.
- The compiler will create this stack.

7.10 When there is no need for a stack

- Sometimes we don't need to preserve the variables and data structures declared before a function.
- Depending on the compiler, if the function call is the last thing done, the variables are not pushed to the call stack.

7.11 Tail recursion

- The stack saves and preserves the context, meaning variables and data structures for their usage, this is what makes recursion possible.
- Without the stack and the preservation of the current context recursion would not be possible.
- When the recursion function is the last thing performed in the function, we call it *tail recursion*; when there are more instructions below the recursive call we call it *non-tail recursion*, tail is more efficient because the context must not be pushed to the stack, the non-tail recursion is more inefficient for the extra time pushing and popping.

7.12 Recursion versus iteration

This is a debate and every programmer must know this.

7.12.1 When both are equivalent

- In the case of tail recursion, you can do this using a for loop, we know that tail recursion doesn't have a stack in memory, so this is equivalent for tail recursion.

Be aware of apparent tail recursion that looks like tail recursion but is not.

```c
long factorial(unsigned n){
    if (n == 0){
        return 1L;
    }
    return n * factorial(n-1);
    // after the recursive call is done
    // we multiply n, this implies a stack being created in memory.
}
```

7.12.2 When a loop is better

- Use a loop when you can not find an equivalent of recursive logic to implement tail recursion.
- Tail recursion takes as much complexity as a for loop, thus when we can not find an equivalent tail-recursive operation, we must go with a for loop.

7.12.3 When recursion is better

Take the example of a decimal to binary converter, without the stack this problem cannot be solved, we can use the stack provided by the compiler, the alternative implementation to the recursion is iteration, in the iteration we need to implement the stack ourselves, keep track of all the variables, this implies more code, less readability and bigger code. Better we use the stack already implemented for us and use recursion. In conclusion when you need a stack and the stack is indispensable to the purpose, use the call stack and do the job recursively, if you don't need a stack, don't use a stack.

7.12.4 Synthesis in a programmer's way

```
1 if you can convert the recursion to tail recursion then
2     Iteration and recursion are equivalent;
3     # the factorial implementation the call stack is required for the recursive
        implementation, but you can do the same without a stack, for factorial the
        iteration is better.
4 if we can not convert the loop to tail recursion && it doesn't require a stack then
5     iteration is better;
6 if when we need a stack explicitly to solve a problem then
7     recursion is better;
8     # This is preferred rather to implementing our own stack.
```

Algorithm 1: When to use recursion or iteration