Fundamental Data Structures & Algorithms using the C language

David Corzo

2020 May 18

Contents

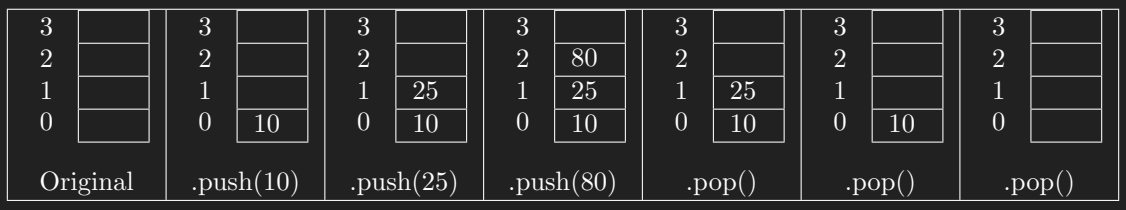
Chapter 1

All about the stack

1.1 Introduction to the stack

- The stack is a linear data structure where the elements could be added or deleted only at one open end called the top of the stack.
- The elements follow "Last in first out", typically called LIFO.
- Insertion into the stack is called "push" operation and deletion from the stack is called the "pop" operation.
  - If you need to add an element to the stack, you need to call the push operation.
  - If you need to delete an element of the stack, you need to call the pop operation.
- If the stack is full it is said to be in "Overflow" state, and push is rejected if stack is overflowed.
- If the stack is empty it is said to be in "Underflow" state, and pop is rejected if the stack is underflowed.

1.1.1 Example of the stack and the push, pop method



1.2 Practical examples where stacks are used

- Text editors use a stack when registering new words and undoing one with ctrl+z. In the example below:

```
stack.push("First");
stack.push("Second");
stack.push("Third");
```

```
First
Second
Third
```

```
// ctrl+z in keyboard
stack.pop();
```

```
First
Second
```

```
stack.pop();
```

```
First
```

- The back button in the browser is also a stack implementation.
- In the command line, the terminal saves the commands executed in a stack, the upper arrow key access the commands. The commands are stored in a stack.
- There are plenty of examples.

1.3 Basic algorithm for stack data structure

- Implementation of stack using 1-D array.

```
int stack[5];
int index = -1;
```

3
2
1
0
 	
Stack is empty or underflowed.
	3
	2
	1
	0 10
	
	++index;
	stack[index] = 10;
	//index:0
		3
		2
		1 25
		0 10
		
		++index;
		stack[index] = 25;
		//index:1
			3
			2 80
			1 25
			0 10
			
			++index;
			stack[index] = 80;

3 2 1 25 0 10 stack[index] = 0; index--;	3 2 1 0 10 stack[index] = 0; index--;	3 2 1 0 stack[index] = 0; index--;	

- Push pseudo-code:

```
1 initialize(stack);
2 if stack.top ≠ stack.size - 1 then
3 |   display "STACK OVERFLOW";
4 |   exit push;
5 end
6 stack.top = stack.top + 1;          ▷ Increment by one;
7 stack[stack.top] = v;               ▷ Assign element at the top position;
8 end procedure push;
```

Algorithm 1: Procedure Push (v)

- Pop pseudo-code:

```
1 if stack.top == -1 then
2 |   display "STACK OVERFLOW";
3 |   exit pop;
4 end
5 v = stack[stack.top];
6 stack.top = stack.top - 1;
7 return v;
8 end procedure POP;
```

Algorithm 2: Procedure pop ()

1.4 Implementation of stack - Code along

- After including the standard libraries:

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

- Lets implement a typedef struct with a fixed size member array called item.

```c
#define SIZE 10
typedef struct {
    int item[SIZE];
    int top;
} Stack;
```

- Now the functions pertinent to the stack: push, pop.

```c
// Prototype declarations
void push(Stack *, int );
int pop(Stack *);
```

```c
void push(Stack *sp, int value ){
    if (sp->top == SIZE -1){
        perror("Stack overflow.\n");
        exit(-1);
    }
    ++sp->top;
    sp->item[sp->top] = value;
}
int pop(Stack *sp){
    if (sp->top == -1){
        printf("Stack underflow.\n");
        return -9999; // this will be the error code.
    }
    --sp->top; // don't mind
    return sp->item[sp->top + 1];
}
```

- For functionality we'll implement some utility functions called init, print_s:
  - init(): initializes the top struct stack member to -1.
  - print_s(): prints the used members of the stack array (item).

```c
// protype declarations
void init(Stack *);
void print_s(Stack *, const char[]);
```

```c
void init(Stack *sp){
    sp->top = -1;
}
void print_s(Stack *sp, const char str[]){
    printf("%s:\n",str);
    for (int i = 0; i < sp->top + 1; i++) {
        printf("\t%d\n",sp->item[i]);
    }
}
```

- Now we'll implement the main function and call every function that we have declared thus far.

```c
int main() {
    Stack s1 = {}, s2 = {}; // initialize everything to 0
    init(&s1);
    init(&s2);
    // push to s1
    push(&s1,100);
    push(&s1,200);
    push(&s1,356);
    push(&s1,120);
    // push to s2
    push(&s2,189);
    push(&s2,167);
    push(&s2,156);
    push(&s2,789);

    print_s(&s1,"s1"); // print used elements in s1.item

    // pop operation
    printf("deleted from s1: %d\n\n",pop(&s1));
    printf("deleted from s1: %d\n\n",pop(&s1));
    printf("deleted from s1: %d\n\n",pop(&s1));

    print_s(&s1,"s1"); // print used elements in s1.item

    printf("--------------------------\n");

    print_s(&s2,"s2"); // print used elements in s2.item

    // pop operation
    printf("deleted from s2: %d\n\n",pop(&s2));
    printf("deleted from s2: %d\n\n",pop(&s2));
    printf("deleted from s2: %d\n\n",pop(&s2));

```

```
    print_s(&s2,"s2"); // print used elements in s1.item
    return 0;
}
```

- The results are as follows:

```
/* Output:
s1:
    100
    200
    356
    120
deleted from s1: 120

deleted from s1: 356

deleted from s1: 200

s1:
    100
-------------------------
s2:
    189
    167
    156
    789
deleted from s2: 789

deleted from s2: 156

deleted from s2: 167

s2:
    189

*/
```

1.5 Making a menu for the stack

- To add a menu to the stack implementation previously demonstrated, apart from the libraries already included, include the stdbool.h header file:

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

- The struct Stack definition remains the same, as well as the the functions: print_s, init, pop, push.
- The main function gets a menu added.
  - We'll add a switch statement for menu purposes.

```
int main() {
    Stack s1 = {};
```

```
    init(&s1);

    printf("1. Push\n2. Pop\n3. Print Stack\n4. Exit\n\n");
    int choice, value;

    while (true){
        printf("Enter choice:");
        scanf("%d",&choice);
        switch (choice) {
        case 1:
            printf("Enter integer value: ");
            scanf("%d",&value);
            push(&s1,value);
            printf("\n");
            break;
        case 2:
            value = pop(&s1);
            if (value != -9999){
                printf("Poped data: %d\n", value);
            }
            break;
        case 3:
            print_s(&s1,"s1");
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice.\n");
            break;
        }
    }
    return 0;
}
```

- An exemplary result is:

```
/* Output:
1. Push
2. Pop
3. Print Stack
4. Exit

Enter choice:1
Enter integer value: 123131

Enter choice:1
Enter integer value: 3423

Enter choice:1
Enter integer value: 234234

Enter choice:1
Enter integer value: 23425

```

```
Enter choice:1
Enter integer value: 890

Enter choice:3
s1:
    [0]: 123131
    [1]: 3423
    [2]: 234234
    [3]: 23425
    [4]: 890
Enter choice:2
Poped data: 890
Enter choice:2
Poped data: 23425
Enter choice:2
Poped data: 234234
Enter choice:2
Poped data: 3423
Enter choice:3
s1:
        [0]: 123131
Enter choice:4

*/
```

1.6 Dynamic memory in the usage of our stack, stack evolution

- For this include the libraries:

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

- The struct Stack member item is a fixed size array, optimally we would like to allocate memory dynamically for this member. Thus, we re-declare the struct Stack member item as a pointer:

```
typedef struct {
    int *item;
    int top;
    int size;
} Stack;
```

- Redefine some functions:

```
// Prototype declarations
void push(Stack *, int );
int pop(Stack *);
void init(Stack *, int);
void print_s(Stack *, const char[]);
void deallocate(Stack *);
```

  - The push function is redefined to include the dynamic memory reallocation in case of stack overflow.

- The pop function stays the same.
- The init function is changed to include memory allocation using malloc() for the member item, and modify the size accordingly.
- The print_s is changed to include printing of members all the way up to the size of memory allocated for the array.
- The deallocate function, frees the space dynamically allocated by the malloc function, also

```c
void push(Stack *sp, int value ){
    if (sp->top == sp->size -1){
        printf("Stack overflow.\n");
        sp->item = (int*)realloc(sp->item,sizeof(sp->size)*2);
        if (sp->item == NULL)
            printf("Error in realloc.\n");
        sp->size *= 2;
    }
    ++sp->top;
    sp->item[sp->top] = value;
}

int pop(Stack *sp){
    if (sp->top == -1){
        printf("Stack underflow.\n");
        return -9999;
    }
    --sp->top; // don't mind
    return sp->item[sp->top + 1];
}

void init(Stack *sp, int size){
    sp->top = -1;
    sp->item = (int*) malloc(size * sizeof(int));
    if (sp->item == NULL){
        printf("Error.\n");
        exit(1);
    }
    sp->size = size;
}

void print_s(Stack *sp, const char str[]){
    printf("%s:\n",str);
    for (int i = 0; i < sp->size; i++) {
        printf("\t[%d]: %d\n",i,sp->item[i]);
    }
}

void deallocate(Stack *sp){
    if (sp->item != NULL)
        free(sp->item);
    sp->top = -1;
    sp->size = 0;
}
```

- The main function changes to be:

– The exit function includes the deallocate function, and the previously redefined push function reallocates the memory every time a stack overflow occurs.

```c
int main() {
    Stack s1 = {};
    init(&s1,3);

    printf("1. Push\n2. Pop\n3. Print Stack\n4. Exit\n\n");
    int choice, value;

    while (true){
        printf("Size of the stack is: %d\n",s1.size);
        printf("Enter choice:");
        scanf("%d",&choice);
        printf("\n");
        switch (choice) {
        case 1:
            printf("Enter integer value: ");
            scanf("%d",&value);
            push(&s1,value);
            printf("\n");
            break;
        case 2:
            value = pop(&s1);
            if (value != -9999){
                printf("Poped data: %d\n", value);
            }
            break;
        case 3:
            print_s(&s1,"s1");
            printf("\n");
            break;
        case 4:
            deallocate(&s1);
            exit(0);
            break;
        default:
            printf("Invalid choice.\n");
            break;
        }
    }
    return 0;
}
```

- All of which prompt the following exemplary results:

```
/* Output:
1. Push
2. Pop
3. Print Stack
4. Exit

Size of the stack is: 3
Enter choice:1
```

```
9
10 Enter integer value: 45
11
12 Size of the stack is: 3
13 Enter choice:1
14
15 Enter integer value: 56
16
17 Size of the stack is: 3
18 Enter choice:1
19
20 Enter integer value: 12
21
22 Size of the stack is: 3
23 Enter choice:1
24
25 Enter integer value: 23
26 Stack overflow.
27
28 Size of the stack is: 6
29 Enter choice:1
30
31 Enter integer value: 25
32
33 Size of the stack is: 6
34 Enter choice:1
35
36 Enter integer value: 14
37
38 Size of the stack is: 6
39 Enter choice:1
40
41 Enter integer value: 36
42 Stack overflow.
43
44 Size of the stack is: 12
45 Enter choice:1
46
47 Enter integer value: 17
48
49 Size of the stack is: 12
50 Enter choice:1
51
52 Enter integer value: 18
53
54 Size of the stack is: 12
55 Enter choice:1
56
57 Enter integer value: 19
58
59 Size of the stack is: 12
60 Enter choice:1
61
62 Enter integer value: 20
```

```
63
64 Size of the stack is: 12
65 Enter choice:3
66
67 s1:
68     [0]: 45
69     [1]: 56
70     [2]: 12
71     [3]: 23
72     [4]: 25
73     [5]: 14
74     [6]: 36
75     [7]: 17
76     [8]: 18
77     [9]: 19
78     [10]: 20
79     [11]: 0
80
81 Size of the stack is: 12
82 Enter choice:2
83
84 Poped data: 20
85 Size of the stack is: 12
86 Enter choice:2
87
88 Poped data: 19
89 Size of the stack is: 12
90 Enter choice:2
91
92 Poped data: 18
93 Size of the stack is: 12
94 Enter choice:2
95
96 Poped data: 17
97 Size of the stack is: 12
98 Enter choice:3
99 */
```

1.7 Stack in action - Decimal to binary conversion

1.7.1 Decimal to binary conversion

- Let's say we want to convert 10 to a binary representation.

$$10 \rightarrow 10 + \frac{0}{2} : (10 \pmod 2) = 0 \text{ LSB}$$
$$10/2 \rightarrow 5 + \frac{0}{2} : (5 \pmod 2) = 1$$
$$5/2 \rightarrow 2 + \frac{1}{2} : (2 \pmod 2) = 0$$
$$2/2 \rightarrow 1 + \frac{0}{2} : (1 \pmod 2) = 1 \text{ MSB}$$

1.7.2 Let's implement the function printBinary

- Taking in to account the process observed above.

```c
// function prototype
void printBinary(unsigned int);
```

```c
void printBinary(unsigned int num){
    Stack s;
    const int base = 2;
    const int num_ = num;
    init(&s,10);
    int rem;

    while (num > 0){
        rem = num % base;
        push(&s,rem);
        num /= base;
    }
    printf("Binary equivalent of %d is: \n", num_);
    while (s.top != -1){
        printf("%d", pop(&s));
    }
    deallocate(&s);
}
```

- Defining a test main function we have:

```c
int main() {
    printBinary(10);
    return 0;
}
```

- Of which results are:

```c
/* Output:
Binary equivalent of 10 is:
1010
*/
```

1.8 Stack in action: Reversing the content of a text file

- Data structure needed: stack of characters.
- Algorithm:
    1. Go on reading characters from source file until End-Of-File is reached.
    2. PUSH each character read into the stack.
    3. When done, POP characters from the stack and write them into the destination file until stack is underflow.

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

```c
#include <stdbool.h>

typedef struct {
    char *item;
    int top;
    int size;
} Stack;

// Prototype declarations
void push(Stack *, char );
char pop(Stack *);
void init(Stack *, int);
void print_s(Stack *);
void deallocate(Stack *);
bool isUnderFlow(Stack *);
bool isOverFlow(Stack *);
int reverse(char[], char[]);


int main()
{
    reverse("data.txt","data_inverse.txt");
    return 0;
}

int reverse(char src[], char dest[]){
    FILE *fp = NULL;
    Stack sp;

    fp = fopen(src,"r");
    if (fp == NULL){
        printf("Error opening file %s.",src);
        return 0;
    }

    // figuring out the number of chars in the file.
    fseek(fp,0,SEEK_END);
    const int filesize = ftell(fp);
    rewind(fp); // setting fp to the beggining.

    // initializing sp
    init(&sp,filesize+1);

    char c = fgetc(fp);
    while (!feof(fp)){
        push(&sp,c);
        c = fgetc(fp);
    }

    fclose(fp);

    // opening new file to paste the inverted data to the next file.
    fp = fopen(dest,"w+");
    if (fp == NULL){
```

```
57          printf("Error opening file %s.",dest);
58          return 0;
59      }
60
61      while (!isUnderFlow(&sp)){
62          fputc(pop(&sp),fp);
63      }
64
65      fclose(fp);
66      deallocate(&sp);
67      fp = NULL;
68      return 1;
69  }
70
71  bool isUnderFlow(Stack *sp){
72      return sp->top == -1;
73  }
74
75  bool isOverFlow(Stack *sp){
76      return sp->top == sp->size - 1;
77  }
78
79  void push(Stack *sp, char value ){
80      if (sp->top == sp->size -1){
81          printf("Stack overflow.\n");
82          sp->item = (char*)realloc(sp->item,sizeof(sp->size)*2);
83          if (sp->item == NULL)
84              printf("Error in realloc.\n");
85          sp->size *= 2;
86      }
87      ++sp->top;
88      sp->item[sp->top] = value;
89  }
90
91  char pop(Stack *sp){
92      if (sp->top == -1){
93          printf("Stack underflow.\n");
94          return '\0';
95      }
96      --sp->top;
97      return sp->item[sp->top + 1];
98  }
99
100 void init(Stack *sp, int size){
101     sp->top = -1;
102     sp->item = (char*) malloc(size * sizeof(char));
103     if (sp->item == NULL){
104         printf("Error.\n");
105         exit(1);
106     }
107     sp->size = size;
108 }
109
110 void print_s(Stack *sp){
```

```c
    for (int i = 0; i < sp->top; i++) {
        printf("%c",sp->item[i]);
    }
}

void deallocate(Stack *sp){
    if (sp->item != NULL)
        free(sp->item);
    sp->top = -1;
    sp->size = 0;
}
/* Input: in file
Hello World!

*/
/* Output:

!dlroW olleH
*/
```

Chapter 2

Step-by-step developing a parenthesis checking program

2.1 Understanding the problem

- To establish a precedence in mathematical operations, we use brackets, the program must check if the program's input has well-formed parenthesis and braces:

$$[\{(a+b)\}/k] \text{ Correct}$$
$$)a+b(\text{ Incorrect}$$
$$\{(a+b)/c] \text{ Incorrect}$$

- For checking this we need a stack.
- The stack comes in at: whenever you are getting an opening bracket in an expression while scanning the expression from left to right, push the bracket into the stack, and whenever you are pushing the closing bracket, check whether the top of the stack contains the opening match for this closing bracket or not.
  - In the expression $\{(a+b)/c]$ the { will be closed first.
- What we'll do while scanning the expression, if we encounter an opening bracket we'll pushing it to the stack, whenever we encounter a closing bracket, we'll check if the top of the stack contains the opening bracket or not, if the top of the stack contains the opening match for this particular closing bracket, we'll pop the stack, and then continue scanning doing the same thing if we encounter another closing or opening bracket.
- If the expression encounters a closing bracket and the stack is empty the that is an error.

2.1.1 Example

- Given a string s=$[2*3\{(3-1)/(5+7)\}-4]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
[	2	*	3	{	(	3	-	1	)	/	(	5	+	7	)	}	-	4	]

```
1 Stack sp;
```

- When scanning position 1, we push the opening brace to the stack.

```
push(&sp,s[0]);
```

- Again when position 4, we push:

```
push(&sp,s[4]);
```

- Again at position 5:

```
push(&sp,s[5]);
```

- The stack looks like this.

2	(
1	{
0	[

- When encountering the closing parenthesis at position 9 we check if the top of the stack is an opening parenthesis. It indeed is, thus we pop the parenthesis from the stack.

```
pop(&sp); // pops (
```

- Encountering position 11, we push the brack to the stack.

```
push(&sp,s[11]);
```

- Encountering position 15, we check the top of the stack for an opening parenthesis, then we pop.

```
pop(&sp); // pops (
```

- Encountering closing } on position 16, we check if the top is {, then pop if it is.

```
pop(&sp); // pops {
```

- Encountering position 19, we check the top for [then we pop if it is that.
- The stack is in underflow now.

2.1.2 Errors

- In the above example no errors exits, an error will occur when a closing bracket does not match an opening bracket in the stack, or when a closing bracket is encountered with the stack being empty.
- Given a string s=$3 * [(8 + 9)/9 - 2\}/5$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	*	[	(	8	+	9	)	/	9	-	2	}	/	5

- The position 12 will cause an error because a closing brace is not matched by an opening brace at the top of the stack.

2.2 Developing the algorithm for bracket checking

Pseudo-code:

```
error = false
while Not end of the expression:
    next_char = next character of input expression;
    if next_char == '(' OR next_char == '{' or next_char == '[' then:
        push(STACK,next_char)
    else if next_char == ')' OR next_char == '}' OR next_char == ']' then:
        if isEmpty(STACK) then:
            error = TRUE
            break while
        else if isOpenningMatch(stacktop(STACK), next_char) then:
            pop(STACK)
        else
            error = TRUE
            break while
        end if
    end if
end while

if !error and !isEmpty(STACK) then:
    error = TRUE
end if

if error then:
    print "The input expression does not contain well formed brackets."
else:
    print "The input expression is well formed."
end if
```

- Without the stack this had not been able to be implemented as easily as it did.

2.3 Implementation of parenthesis checking program

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

#define SIZE 100

typedef struct {
    char item[SIZE];
    int top;
} Stack;

// function prototypes
void push(Stack *, char);
char pop(Stack *);
int isEmpty(Stack *);
char stacktop(Stack *);
```

```
18
19 // utility function prototypes
20 int isOpenMatch(char, char);
21
22 // the bracket checking program function protoype
23 bool checkExpr(char[]);
24 void print_s(Stack *);
25
26 void push(Stack *sp, char c){
27     if (sp->top == SIZE - 1){
28         printf("Stack overflow.\n");
29         exit(1);
30     }
31     sp->top += 1;
32     sp->item[sp->top] = c;
33 }
34
35
36 char pop(Stack *sp){
37     if (sp->top == -1){
38         printf("Stack underflow");
39         exit(1);
40     }
41     return sp->item[sp->top--];
42 }
43
44
45 int isEmpty(Stack *sp){
46     return sp->top == -1;
47 }
48
49
50 char stacktop(Stack *sp){
51     if (isEmpty(sp)){
52         printf("Stack underflow, terminating program.\n");
53         exit(1);
54     }
55     return sp->item[sp->top];
56 }
57
58 int isOpenMatch(char opening, char closing){
59     bool match = false;
60     switch (opening) {
61         case '(':
62             if (closing == ')') {
63                 match = true;
64             }
65             break;
66
67         case '{':
68             if (closing == '}'){
69                 match = true;
70             }
71             break;
```

```
        case '[':
            if (closing == ']'){
                match = true;
            }
            break;
        default:
            printf("Invalid symbol, program is terminating.");
            exit(1);
    }
    return match;
}

void print_s(Stack *sp){
    for (int i = 0; i < sp->top; i++){
        printf("stack: %c\n",sp->item[i]);
    }
}

bool checkExpr(char str[]){
    bool error = false;
    Stack s = {.top=-1};

    int i = 0;
    while (str[i] != '\0'){
        if ( (str[i] == '(') || (str[i] == '{') || (str[i] == '[')){
            push(&s,str[i]);
        } else if ( (str[i] == ')') || (str[i] == '}') || (str[i] == ']')){
            if (isEmpty(&s)) {
                error = true;
                break;
            } else if (isOpenMatch(stacktop(&s),str[i])) {
                pop(&s);
            } else {
                error = true;
                break;
            }
        }
        i++;
    }

    if (!error && !isEmpty(&s)) {
        error = true;
    }

    return error;
}


int main() {
    char expr[SIZE];
    printf("Input expression: ");
    scanf("%[^\n]%*c",expr);

    printf("\n%s\n",expr);
```

```
126
127     checkExpr(expr)? printf("Not well formed.\n") : printf("Well formed.\n");
128
129     return 0;
130 }
131
132 /* Output:
133 Input expression: {()[]}
134
135 {()[]}
136 Well formed.
137
138 */
```

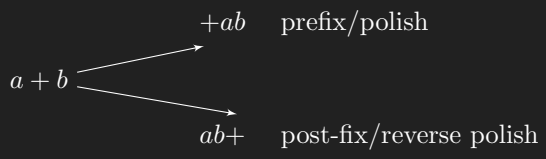
Chapter 3

Polish notation and reverse polish notation

3.1 Polish & Reverse polish notations

3.1.1 Polish notation

- In order to evaluate expressions. For example $(a+b)*c$.
  - We must first understand precedence, which means, the parenthesis are done first because it has a higher precedence, what ever is inside the parenthesis will be done first and then multiplied by c.
- There are a handful of notations, the above example $(a+b)*c$ is called the infix notation.
  - There are other notations such as polish and reverse polish notation.
- In order to convert $a+b$ notation to polish notation:



- For evaluating infix expressions complexity tends to rise, since you need to keep details like what is being evaluated and so on, this complexity diminishes with the prefix and post fix notations. This is why we must aspire to make a converter of infix to prefix or post fix notation.
  - This is beneficial and reduces complexity.
- Example: given the infix expression $(a+b)*c$ find the equivalent in terms of a prefix expression.

$$\overbrace{(a+b)}^{+ab}*c = \overbrace{+ab}^{r}*c = r*c = *rc = *+abc$$

  - $*+abc$ this is going to be read from right to left, the first operator is going to be $+$ this allows precedence and allows for no use of braces.

3.1.2 Reverse polish notation

- In order to convert from infix to polish notation using the example $(a+b)*c$:

$$\overbrace{(a+b)}^{ab+}*c = \overbrace{ab+}^{r}*c = r*c = rc* = ab+c*$$

- In the reverse polish notations reading it from left to right the first operation will be $+$.

3.2 Understanding precedence of operators, conversion idea - infix to prefix/post-fix

- Let \$ be a substitute for the character ^, to represent exponential operations.
- Remember the precedence of operators go as follows:

↑	()
↑	*, /
↑	+, -

- Given the following expression: $a+b\$(c-d)k\$p$ convert to polish notation.

$$a+b\$(c-d)k\$p = a*b\$-cd/k\$p = a*\$-cd/k\$p = a*\overbrace{\$b-cd}^{r_1}/\overbrace{\$kp}^{r_2}$$
$$= a*r_1/r_2 = *ar_1/r_2 = *ar_1/r_2 = /*ar_1r_2 = */a\$b-cd\$kp$$

  - The equivalent expression in prefix notation reading from right to left $*/a\$b-cd\$kp$
- Given the same expression convert it to post-fix.

$$a+b\$(c-d)k\$p = a*b\$cd-/k\$p = a*bcd-\$/kp\$ = abcd-\$*/kp\$ = abcd-\$*kp\$/$$

3.3 How to evaluate polish or reverse polish notation

- Iterating a string `"24+32-*"` written in reverse polish notation. Program to implement has to iterate and will for this example only work for single digit numbers.
- Reverse polish notation, remember, is read from left to right.

0	1	2	3	4	5	6
2	4	+	3	2	-	*

- Position 0 and 1 are operands and must be pushed to the stack.
- Position 2 is, however, an operator, poping the stack twice will allow us to perform the operation of adding the operands, the result of both is pushed to the stack.
- Position 3 and 4 are operands, the operator on position 5 pops the stack twice, subtracts the operands and pushes the result to the stack.
- The position 6 operand pops the stack twice and performs the operation, pushing the value to the stack, poping the stack one time will reveal the final result.
- For the polish notation start at the end and do the same thing.

3.4 Algorithm for evaluating post-fix expression

```
1  initialize(stack);
2  while not end of the post-fix string do
3  |  next_current = gets the next token from the string;
4  |  if next_token is an operand then
5  |  |  push(stack, next_token);
6  |  else if next_token is an operator then
7  |  |  operand1 = pop(stack);
8  |  |  operand2 = pop(stack);
9  |  |  result = operate(operand2, operand1, next_token);
10 |  |  push(stack, result);
11 end
12 print pop(stack);
13 end procedure interpret post-fix expressions;
```

Algorithm 3: Interpret post-fix expressions pseudo-code

3.5 Implementing evaluation of post-fix expressions with the C programming language

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>
#define SIZE 100

typedef struct {
    double item[SIZE];
    int top;
} OperandStack;


// func prototypes

// func for stack operations
void push(OperandStack *, double);
double pop(OperandStack *);
int isEmpty(OperandStack *);
char stacktop(OperandStack *);

// utility function prototypes
int isOpenMatch(char, char);
double operate(double, double, char);
bool isOperator(char c);
double evalPostfix(char[]);
void print_s(OperandStack *);


void push(OperandStack *sp, double c){
```

```
30      if (sp->top == SIZE - 1){
31          printf("Stack overflow.\n");
32          exit(1);
33      }
34      sp->top += 1;
35      sp->item[sp->top] = c;
36  }
37
38
39  double pop(OperandStack *sp){
40      if (sp->top == -1){
41          printf("Stack underflow");
42          exit(1);
43      }
44      return sp->item[sp->top--];
45  }
46
47
48  int isEmpty(OperandStack *sp){
49      return sp->top == -1;
50  }
51
52
53  char stacktop(OperandStack *sp){
54      if (isEmpty(sp)){
55          printf("Stack underflow, terminating program.\n");
56          exit(1);
57      }
58      return sp->item[sp->top];
59  }
60
61  int isOpenMatch(char opening, char closing){
62      bool match = false;
63      switch (opening) {
64          case '(':
65              if (closing == ')') {
66                  match = true;
67              }
68              break;
69
70          case '{':
71              if (closing == '}'){
72                  match = true;
73              }
74              break;
75          case '[':
76              if (closing == ']'){
77                  match = true;
78              }
79              break;
80          default:
81              printf("Invalid symbol, program is terminating.");
82              exit(1);
83      }
```

```
84      return match;
85  }
86
87  double operate(double left, double right, char operator){
88      double result = 0;
89      switch (operator){
90          case '+': result = left + right;
91                    break;
92          case '-': result = left - right;
93                    break;
94          case '*': result = left * right;
95                    break;
96          case '/': result = left / right;
97                    break;
98          case '$': result = pow(left, right);
99                    break;
100         default: printf("%c is not a valid operator\n",operator);
101                   exit(1);
102     }
103     return result;
104 }
105
106 bool isOperator(char c){
107     return  (c == '+') ||
108             (c == '-') ||
109             (c == '*') ||
110             (c == '/') ||
111             (c == '$');
112 }
113
114 double evalPostfix(char postfix[]){
115     OperandStack stack = {.top=-1};
116     int i = 0;
117     while (postfix[i] != '\0'){
118         char token = postfix[i];
119         if (token >= '0' && token <= '9'){
120             int v = token - '0'; // you can also use a atoi func
121             push(&stack,(double)v);
122         } else if (isOperator(token)){
123             double right = pop(&stack);
124             double left = pop(&stack);
125             double result = operate(left, right, token);
126             push(&stack,result);
127         } else {
128             printf("Invalid symbol encontered \"%c\"",token);
129             exit(1);
130         }
131         ++i;
132     }
133     return pop(&stack);
134 }
135
136 void print_s(OperandStack *sp){
137     for (int i = 0; i < sp->top; i++){
```

```
        printf("stack: %c\n",sp->item[i]);
    }
}

int main() {
    char postfix[SIZE];
    printf("Input postfix expression: ");
    scanf("%[^\n]%*c",postfix);

    printf("Result is: %lf", evalPostfix(postfix));

    return 0;
}

/* Output:
Input postfix expression: 45+
Result is: 9.000000
*/
/* Output:
Input postfix expression: 45+36-*
Result is: -27.000000
*/
```

3.6 Understanding the precedence function

- The precedence function takes two arguments, the first argument is the stack top, the second is the token, the function returns true if the first argument (stack top) has equal or higher precedence than the token.
- The function will return false if the token has higher precedence than the second.
- The following rules apply here: (op being any operator encountered)
  - Any operator existing in the stack compared with the `(` character will return false, meaning the `(` has higher precedence than any other operator.

    ```
    precedence(stack_top,'('); // false
    ```

  - If the stack top is `(` character, any operator stored in token has higher precedence, thus the function will return false.

    ```
    precedence('(',token); // false
    ```

  - If the precedence function has a token or second argument equal to the character `)`, this means which ever character stored in the stack top has a higher precedence than closing parenthesis, except the case when the stack top is `(` opening parenthesis.

    ```
    precedence(stack_top,'('); // true
    // except: stack_top = '('
    ```

  - If the stack top is `(` opening parenthesis, and the token is `)` closing parenthesis, this is false.

    ```
    precedence('(',')'); // false
    ```

- Remember the order of operations or precedence is as follows in mathematics:
  1. Anything within parenthesis.

2. Exponentiation and root extraction.
3. Multiplication and division.
4. Addition and subtraction.

- For all intents and purposes for this program if the precedence is the same the precedence function will return true as if one were greater than the other. The following examples apply:

False scenarios	True scenarios
`precedence('+','/') = false;`	`precedence('+','+') = true;`
`precedence('+','$') = false;`	`precedence('+','-') = true;`
`precedence('+','(') = false;`	`precedence('+',')') = true;`
`precedence('-','*') = false;`	`precedence('-','+') = true;`
`precedence('-','/') = false;`	`precedence('-','-') = true;`
`precedence('-','$') = false;`	`precedence('-',')') = true;`
`precedence('-','(') = false;`	`precedence('*','+') = true;`
`precedence('*','$') = false;`	`precedence('*','-') = true;`
`precedence('*','(') = false;`	`precedence('*','*') = true;`
`precedence('/','$') = false;`	`precedence('*','/') = true;`
`precedence('/','(') = false;`	`precedence('*',')') = true;`
`precedence('$','(') = false;`	`precedence('/','+') = true;`
`precedence('(','+') = false;`	`precedence('/','-') = true;`
`precedence('(','-') = false;`	`precedence('/','*') = true;`
`precedence('(','*') = false;`	`precedence('/','/') = true;`
`precedence('(','/') = false;`	`precedence('/',')') = true;`
`precedence('(','$') = false;`	`precedence('$','+') = true;`
`precedence('(','(') = false;`	`precedence('$','-') = true;`
`precedence('(',')') = false;`	`precedence('$','*') = true;`
`precedence(')','+') = false;`	`precedence('$','/') = true;`
`precedence(')','-') = false;`	`precedence('$','$') = true;`
`precedence(')','*') = false;`	`precedence('$',')') = true;`
`precedence(')','/') = false;`	`precedence(')',')') = true;`
`precedence(')','$') = false;`	
`precedence(')','(') = false;`	

3.7 Explaining how it works

- Given the expression $m + (a + ((b - c) * (d + k)))\$(x + y) * p$; its worth noting that this expression can be interpreted as follows:

$$
\begin{aligned}
&= m + \overbrace{(a + ((b - c) * (d + k)))}^{r} \$ \overbrace{(x + y)}^{w} * p \\
&= m + r^{w} * p
\end{aligned}
$$

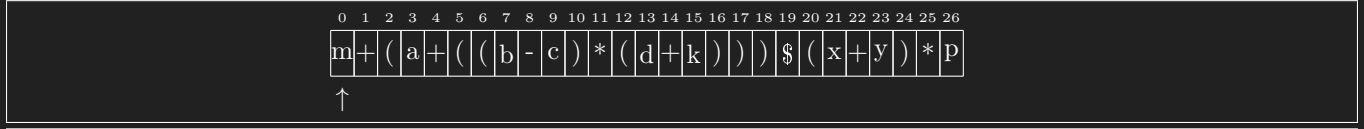
- Let's index the string:

`infix_string =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p

- We'll next iterate throughout the entire length of the string, using a loop and a counter variable.

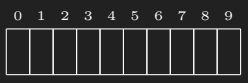
3.7.1 Each index in the example




```
1 infix_string[i=0] = 'm';
```

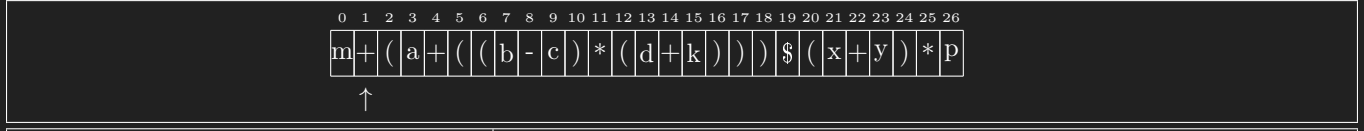
- `'m'` is an operand, thus we just append it to the post-fix string.

- The stack looks like this:



- The post-fix string looks like this:

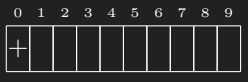
```
1 postfix = "m";
```




```
1 infix_string[i=1] = '+';
```

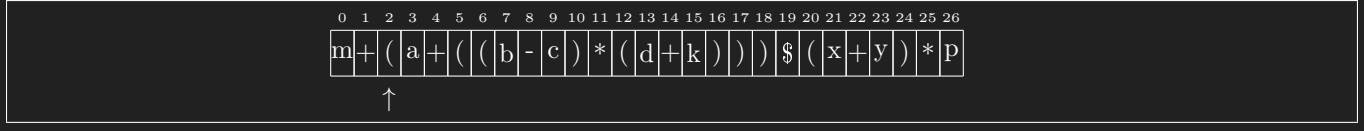
- `'+'` is an operator, since the stack is empty then we just append it to the stack without checking precedence.

- The stack looks like this:



- The post-fix string looks like this:

```
1 postfix = "m";
```



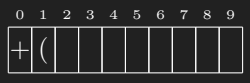
```
infix_string[i=2] = '(';
```

- `'('` is an operator; the stack is not empty so we need to check precedence, taking in to account that the stack top is `'+'` and the token is `'('` if we check the precedence of this using the table above we can observe that the `(` character .
- We push it to the stack if the precedence function evaluates to false.

```
precedence('+','('); // ->
↪  false
```

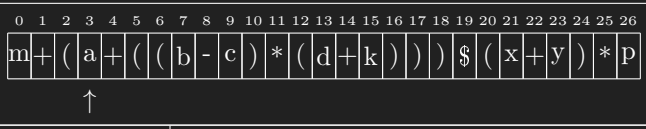
- The stack looks like this:

OperandStack =



- The post-fix string looks like this:

```
postfix = "m";
```




```
infix_string[i=3] = 'a'
```

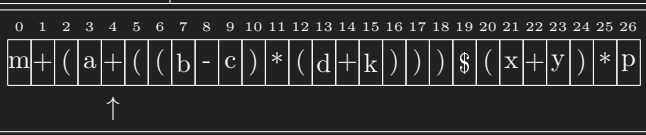
- `'a'` is an operand; this we just append to the post-fix string.

- The stack looks like this:

OperandStack =

- The post-fix string looks like this:

```
postfix = "ma";
```



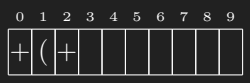

```
infix_string[i=4] = '+'
```

- `'+'` is an operator, the stack is not empty, since this is the case the precedence needs to be evaluated.

```
precedence('(','+'); // ->
↪  false
```

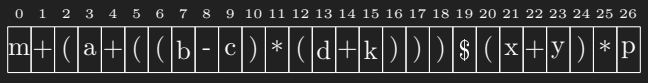
- The stack looks like this:

OperandStack =



- The post-fix string looks like this:

```
postfix = "ma";
```

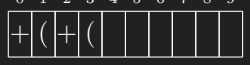



```
infix_string[i=5] = '(';
```

- The parenthesis is an operator, we compare it using the precedence function, and we note that the function returns false because `+` has less precedence than `(`.

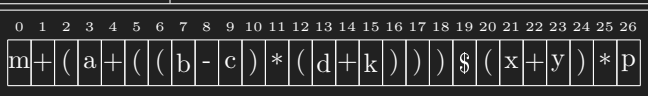
```
precedence('+','('); // ->
↪  false
```

- The stack looks like this:

`OperandStack =` 

- The post-fix string looks like this:

```
postfix = "ma";
```

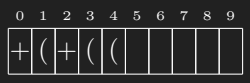



```
infix_string[i=6] = '(';
```

-

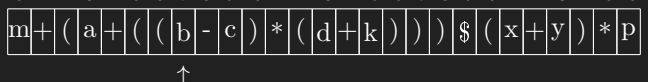
```
precedence('(','('); // ->
↪  false
```

- The stack looks like this:

`OperandStack =` 

- The post-fix string looks like this:

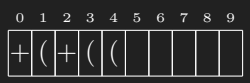
```
postfix = "ma";
```




```
infix_string[i=7] = 'b';
```

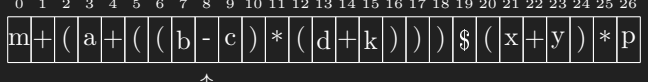
- `'b'` is an operand, so we just append it to the post-fix string.

- The stack looks like this:

`OperandStack =` 

- The post-fix string looks like this:

```
postfix = "mab";
```



```
infix_string[i=8] = '-';
```

- `'-'` is an operator, since the stack is not empty, we call the precedence function, the stack top is `'('` and the token is `'-'`, this evaluates to false.

```
precedence('(','-'); // ->
↪  false
```

- The stack looks like this:

`OperandStack =`

0	1	2	3	4	5	6	7	8	9
+	(	+	(	(	-				

- The post-fix string looks like this:

```
postfix = "mab";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
								↑																		

```
infix_string[i=9] = 'c';
```

- `'c'` is an operand, thus we just append it to the post-fix string.

- The stack looks like this:

`OperandStack =`

0	1	2	3	4	5	6	7	8	9
+	(	+	(	(	-				

- The post-fix string looks like this:

```
postfix = "mabc";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
									↑																	

```
infix_string[i=10] = ')';
```

- `')'` is an operator, the stack is not empty, but, upon checking precedence this evaluates to true.
- The rule when the precedence function returns true is to pop the stack top until the precedence is false again.

```
precedence('-',')'); // -> true
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+	(	(	-				

```
postfix.append(pop(&OperandStack));
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+	(	(					

  - Now we must compare the stack top that is now `'('` with the token which is `')'`, in that case what the precedence function returns is false, but the case will be special.

```
// check precedence again
precedence('(',')'); // -> false
```

  - In the case that stack top is `'('` and the token is `')'`, this case evaluates to false but it will be treated differently.
  - What must be done is pop the stack once and discard the token.

```
pop(&OperandStack);
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+	(						

- The post-fix string looks like this:

```
postfix = "mabc-";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
						↑																				

```
infix_string[i=11] = '*';
```

- The `'*'` character has higher precedence, since the precedence function returns false, we push the character to the stack.

```
precedence('(','*'); // ->
↪  false
```

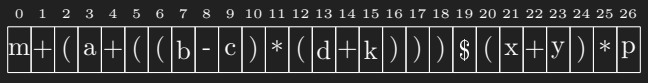
- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+	(	*					

- The post-fix string looks like this:

```
postfix = "mabc-";
```



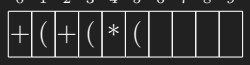

```
infix_string[i=12] = '(';
```

- The character '(' has higher precedence than * thus the precedence function returns false.
- We must push in to the stack.

```
precedence('*','('); // ->
↪  false
```

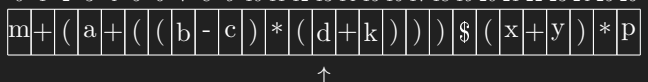
- The stack looks like this:

OperandStack =



- The post-fix string looks like this:

```
postfix = "mabc-";
```

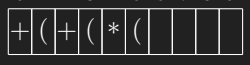



```
infix_string[i=13] = 'd';
```

- This is an operand, we just append to the post-fix string.

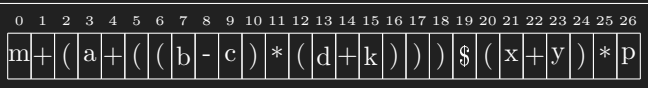
- The stack looks like this:

OperandStack =



- The post-fix string looks like this:

```
postfix = "mabc-d";
```



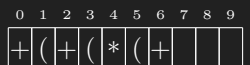

```
infix_string[i=14] = '+';
```

- The '+' has higher precedence, this means the precedence function returns false.

```
precedence('(','+'); // ->
↪  false
```

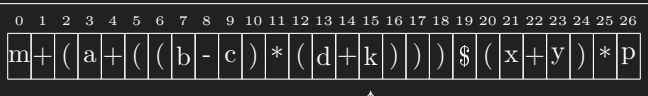
- The stack looks like this:

OperandStack =



- The post-fix string looks like this:

```
postfix = "mabc-d";
```

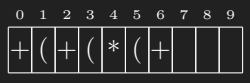


```
infix_string[i=15] = 'k';
```

- `'k'` is an operator, thus we just append it to the post-fix string.

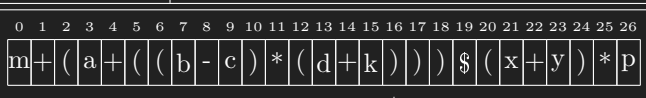
- The stack looks like this:

`OperandStack =`



- The post-fix string looks like this:

```
postfix = "mabc-dk";
```




```
infix_string[i=16] = ')';
```

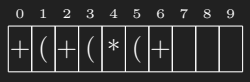
- The stack top is `'+'` and the token is `')'`, meaning that the precedence function will return true.

- This means that we must pop the stack, append the operators poped to the post-fix string until the precedence function returns

```
precedence('+',')'); // -> true
```

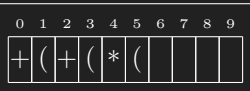
- The stack looks like this:

`OperandStack =`




```
postfix.append(pop(&OperandStack));
```

`OperandStack =`



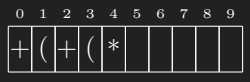
  - Continue with the precedence checking until true.

```
precedence('(',')');
```

  - This is false, but this is the exceptional case, for the case `'('` as the stack top and token `')'` we pop the stack and discard the token as well.

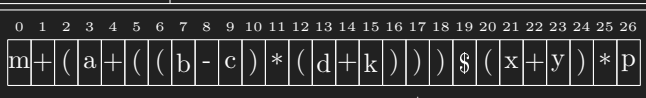
```
pop(&OperandStack);
```

`OperandStack =`



- The post-fix string looks like this:

```
postfix = "mabc-dk+";
```



```
infix_string[i=17] = ')';
```

- This is an operator, and needs to evaluated for precedence, in this case it is true, thus we need to pop the stack and append the poped element to the post-fix string until the precedence function returns false.

```
precedence('*',')'); // -> true
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+	(	*					

  - The `'*'` character constituting the stack top and the token `')'` will cause a true to be returned from the precedence function. Thus pop it and append it to the post-fix string.

```
postfix.append(pop(&OperandStack));
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+	(						

  - The stack top is `'('` and the token is `')'`, this is the exception case, we pop and discard.

```
pop(&OperandStack);
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+							

- The post-fix string looks like this:

```
postfix = "mabc-dk+*";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p

```
infix_string[i=18] = ')';
```

- The stack top is '+' and the token is ')', the precedence function returns true, thus we pop the stack and append to the post-fix string.

```
precedence('+',')'); // -> true
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(	+							

```
postfix.append(pop(&OperandStack));
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	(								

  - Again we check for precedence.

```
precedence('(',')');
```

  - This case we discard.

```
pop(&OperandStack);
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+									

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																		↑								

```
infix_string[i=19] = '$';
```

- The token '$' and the stack top + will cause the precedence function to return false.
- We push the $ to the stack.

```
precedence('+','$'); // -> false
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$								

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																		↑								

```
infix_string[i=20] = '(';
```

- The `'('` has higher precedence, so the function will return false, if it is false we push it to the stack as always.

```
precedence('$','('); // ->
↪  false
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$	(							

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																					↑					

```
infix_string[i=21] = 'x';
```

- `'x'` is an operand. Just append it to the post-fix string.

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$	(							

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+x";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																					↑					

```
infix_string[i=22] = '+';
```

- The `'+'` operator has more precedence than `'('` so the function returns false.
- Thus we just push the token to the stack.

```
precedence('(','+'); // ->
↪  false
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$	(	+						

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+x";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																					↑					

```
infix_string[i=23] = 'y';
```

- This is an operand, we just append it to the post-fix string.

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$	(	+						

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+xy";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																								↑		

```
infix_string[i=24] = ')';
```

- The ')' character has lower precedence than + thus we pop the stack and append the popped item to the post-fix string.

```
precedence('+',')'); // -> true
```

- The stack looks like this:

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$	(	+						

```
postfix.append(pop(&OperandStack));
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$	(							

- Check precedence again.

```
precedence('(',')'); // -> false
```

- This is an exception case, we discard the '('.

```
pop(&OperandStack);
```

OperandStack =

0	1	2	3	4	5	6	7	8	9
+	$								

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+xy+";
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
m	+	(	a	+	(	(	b	-	c	)	*	(	d	+	k	)	)	)	$	(	x	+	y	)	*	p
																								↑		

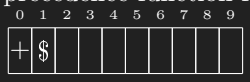
```
infix_string[i=25] = '*';
```

- The '$' returns true because it has higher precedence than '*'.

```
precedence('$','*'); // -> true
```

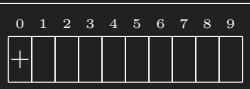
- The stack looks like this:
  - We pop the stack until the precedence function returns true.

OperandStack =




```
postfix.append(pop(&OperandStack));
```

OperandStack =

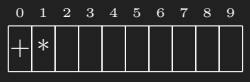


  - We check precedence again.

```
precedence('+','*'); // -> false
```

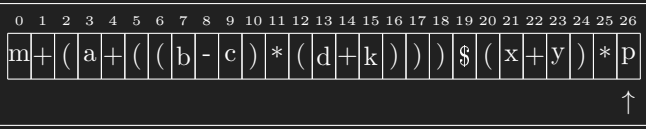
  - Since this is false, we just push it to the stack.

OperandStack =



- The post-fix string looks like this:

```
postfix = "mabc-dk+*+xy+$";
```

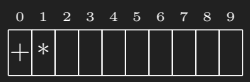



```
infix_string[i=26] = 'p';
```

- This is an operand, thus we just appendt it to the post-fix string.

- The stack looks like this:

OperandStack =



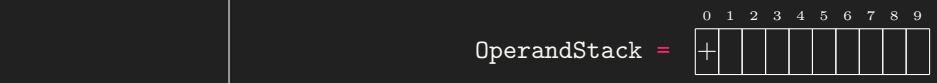
- The post-fix string looks like this:

```
postfix = "mabc-dk+*+xy+$p";
```

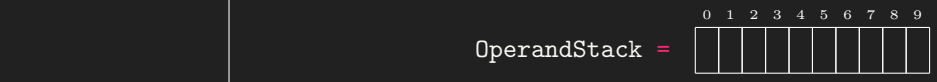
- We have reached the top of the string. Now whatever content is in the stack must be emptyed and appended to the post-fix string.

- The stack looks like this:


```
postfix.append(pop(&OperandStack));
```




```
postfix.append(pop(&OperandStack));
```



  - The stack is now empty.

- The post-fix string looks like this:

```
postfix = "mabc-dk+*+xy+$p*+";
```

3.8 Writing the algorithm for converting infix expression to equivalent post-fix

- Arrays:
  - infix: to hold the infix string.
  - Post-fix: to hold the post-fix string, initially empty.
- Other data structures:
  - OperatorStack: top is initialized and push and pop operation works on this.

Pseudo-code for the infix to post-fix:

```
Initialize;
while not the end of the infix string do
    token = get the next element from infix string;
    if token is an operand then
        append the token with the postfix string;
    else if token is an operator then
        while not empty operator stack and prcd(stacktop,token) do
            top_operator = pop(operatorstack);
            append top_operator with the postfix string;
        end
        if token = ')' then
            pop(operatorstack);
        else
            push(operatorstack,token);
        end
    end
end
while not empty operatorstack do
    top_operator = pop(operatorstack);
    append top_operator with the postfix string;
end
print postfix;
```

Algorithm 4: Algorithm for converting infix to post-fix

3.9 Combine the conversion and evaluation function in a single program

- Now we can combine the program that evaluates the post-fix notation and the infix to post-fix notation.

```c
#include "infixpostfix.h"
#include "evalpostfix.h"

int main()
{
    char infix[100] = "(8+9*7)$(7-4)";
    char postfix[100];

    infix_to_postfix_op(infix,postfix);

    double result = evalPostfix_opa(postfix);

    printf("%s\n",infix);
    printf("%s\n",postfix);
    printf("%lf",result);

    return 0;
}
/* Output:
(8+9*7)$(7-4)
897*+74-$
```

```
357911.000000
*/
```

Chapter 4

All about the queue

4.1 Introduction to the queue

- The queue is similar to the stack, however, the queue follows the first in first out order.
- The queue data structure has two ends, front end and the back end.
- Front end is the end where the elements are deleted.
- Back end is where the elements are inserted.
- FIFO: First In First Out.
- In the stack, we only had une end in which we performed all the operations, we poped from the top, and inserted in the top.
- In the queue we have two ends in which we can perform operations, back end and the front end, back end is for inserting and front end is for deleting.

4.1.1 Formal definition

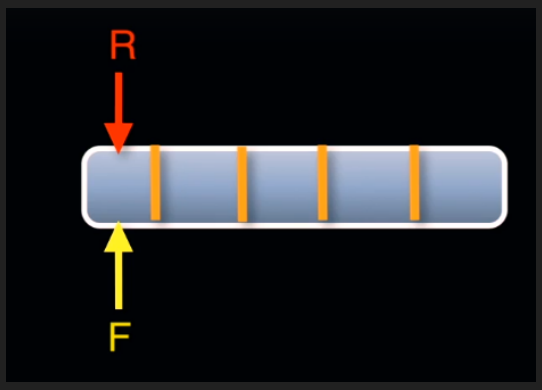
- Queue: Queue is a linear data structure with two open ends, called the "rear" and the "front", elements are added at the "rear" end and deleted from the "front" end.
- The term linear data structure implies that the data structure has no hierarchical order. A tree is a hierarchical data structure, queue is not a hierarchical data structure.
- Elements in a queue typically follows "First In First Out" order, that is elements inserted first will be always deleted first.
- Also called FIFO data structure.

4.1.2 Some other variations of the queue data structure

- Double ended queue: elements can added or deleted form both ends, this means from the rear and the front. It's a mixture of FIFO and LIFO.
- Priority queue: elements are deleted on the basis of predefined priority. In the FIFO queue the priority in deleting is always found in the front. Consider a situation in which an interview is being granted to potential students applying for a job, students are standing in a line following the FIFO order, when the interviewer comes in, he decides he no longer wants the FIFO order and instead wants the highest grade to pass first, this is a priority for the higher grade students. The queue transforms in to a priority queue.
  - This can be implemented using a heap data structure.

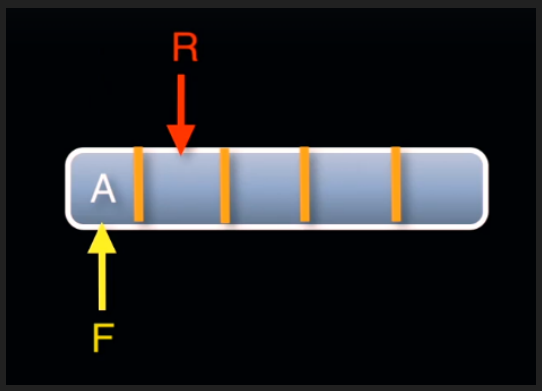
4.2 The FIFO queue implementation idea using array

- We start declaring an array, in the initialization, the rear and the front end will be the same.



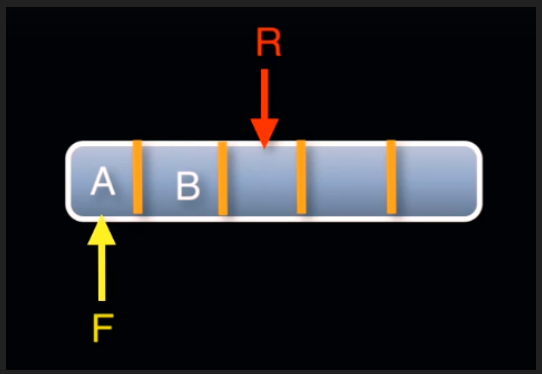
- Now the equivalent of inserting in a queue is called enqueue, and the equivalent of deletion is dequeue. We now enqueue `'A'` to the queue.

```
1 Queue.enqueue('A');
```



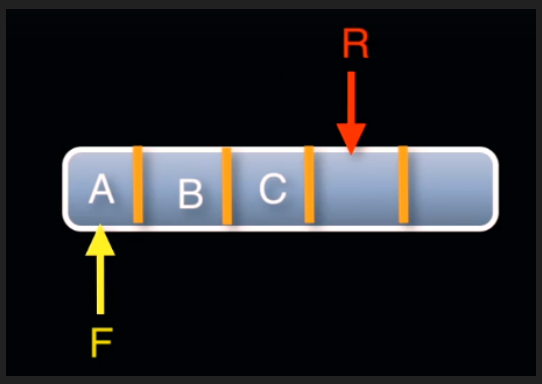
- We enqueue `'B'`.

```
1 Queue.enqueue('B');
```



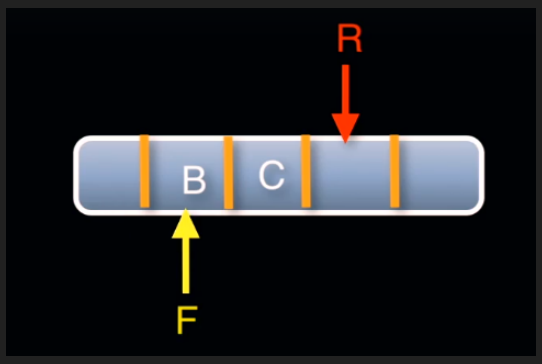
- We enqueue `'C'`.

```
Queue.enqueue('C');
```



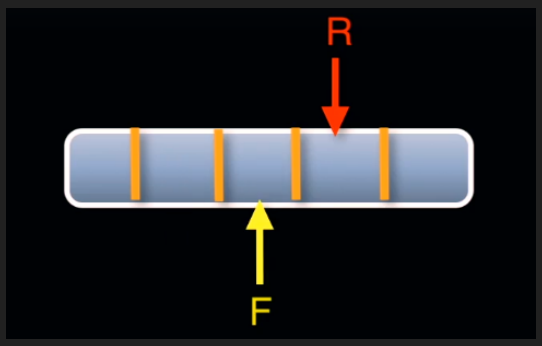
- We now dequeue.

```
Queue.dequeue();
```



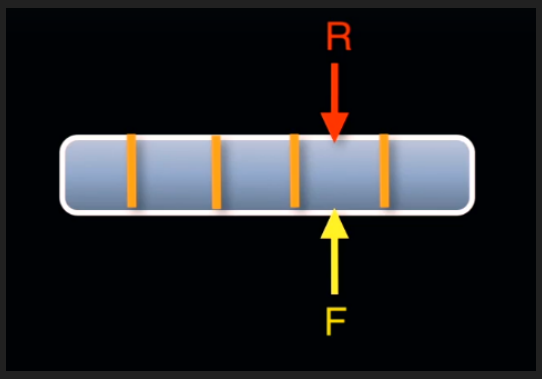
- We dequeue.

```
Queue.dequeue();
```



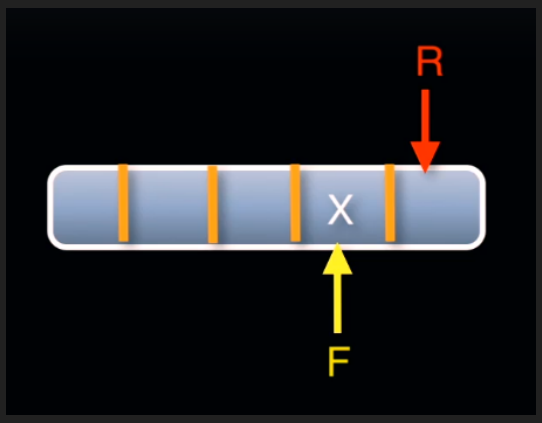
- We are now at an underflow state.

```
Queue.dequeue();
```



- We can enqueue again.

```
Queue.enqueue('X');
```



4.3 Algorithm for FIFO Queue

Initial declarations:

- REAR: is an integer value to hold the index of the rear end of Q, that is the index of the next insertion element.
- FRONT: is an integer variable to hold the index of the front end of Q, that is the index of the next element to be deleted.
- ITEM[SIZE]: is a 1-D array that we will be using for keeping the queue elements SIZE is the size of the queue, that is the number of elements in the array, we consider the index of the array starts from 0.
- Initially:

```
REAR = 0;
FRONT = 0;
```

- Underflow condition: if the rear and front are equal then this the condition for underflow checking, this means the queue is empty.
- Overflow condition: when the rear index goes beyond the last element. When the rear is equal to the size.

- Enqueue: place the new element at the rear index and then move the rear to the next element. If the rear goes beyond the last element that means we have inserted the new element in the last index, we can't insert anymore, this will cause an overflow.
- Dequeue: remove the front element and move the front forward by one.

4.3.1 Enqueue

```
1 OPERATION ENQUEUE (v)
2 if rear == SIZE then
3 |   print "Queue overflow";
4 |   exit queue;
5 end
6 item[rear] = v;
7 rear = rear + 1;
```

Algorithm 5: OPERATION ENQUEUE (v)

4.3.2 Dequeue

```
1 if rear == front then
2 |   print "Queue underflow";
3 |   exit dequeue;
4 end
5 v = item[front];
6 front = front + 1;
7 return v;
```

Algorithm 6: OPERATION DEQUEUE (v)

4.4 Implementation of FIFO Queue

```c
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100

typedef struct {
    int item[SIZE];
    int rear;
    int front;
}Queue;

// function prototypes
int dequeue(Queue *);
void enqueue(Queue *, int);
// functionality functions
void initQueue(Queue *);

void enqueue(Queue *queue, int v){
    if (queue->rear == SIZE){
        printf("Queue overflow\n");
```

```
        exit(1);
    }
    queue->item[queue->rear] = v;
    queue->rear++;
}
int dequeue(Queue *queue){
    if (queue->rear == queue->front){
        printf("Queue underflow\n");
        return -9999;
    }
    int v = queue->item[queue->front++];
    return v;
}
void initQueue(Queue *queue){
    queue->front = queue->rear = 0;
}

int main() {
    Queue q;
    initQueue(&q);
    enqueue(&q, 5);
    enqueue(&q, 10);
    enqueue(&q, 15);
    enqueue(&q, 20);

    printf("%d\n",dequeue(&q));
    printf("%d\n",dequeue(&q));
    printf("%d\n",dequeue(&q));
    printf("%d\n",dequeue(&q));
    printf("%d\n",dequeue(&q));

    return 0;
}
/* Output:
5
10
15
20
Queue underflow
-9999

*/
```

4.5 The loophole in our implementation of FIFO queue

- The queue can't be reused essentially, enqueing 3 elements and then dequeueing them inserts new elements in the third.
- Changing the item memeber size to 5 for practical reasons.

```
#define SIZE 5
```

- We can observe that eventually the queue will be in a state of overflow and underflow at the same time

with the user not being able to do anything to help it.

```
/* Output:
----Queue operations----
1. Enqueue
2. Dequeue
3. Quit
--------------------------
Input your option: 1
Input the value to enqueue: 100
Input your option: 1
Input the value to enqueue: 200
Input your option: 1
Input the value to enqueue: 300
Input your option: 1
Input the value to enqueue: 400
Input your option: 2
Deleted value: 100
Input your option: 2
Deleted value: 200
Input your option: 2
Deleted value: 300
Input your option: 2
Deleted value: 400
Input your option: 2
Queue underflow
Input your option: 1
Input the value to enqueue: 500
Input your option: 1
Input the value to enqueue: 600
Queue overflow
Input your option: 3
*/
```

4.6 Understanding the loophole, why it happens?

- The solution lies in the rear and the front to be relocated to the beginning of the array.
- This is called a circular queue.

4.7 Introduction to circular queue

- We have a queue, instead of moving the front and rear in the way a normal queue operates, we will use the following formulas:

$$r = (r + 1)\% \text{ SIZE}$$

$$p = (p + 1)\% \text{ SIZE}$$

- You can also use if-else statements to do the same thing:

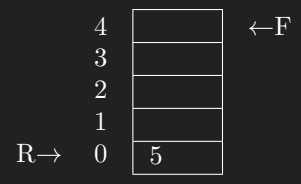
```
1 if rear == size - 1 then
2 |  r = 0;
3 else
4 |  r++;
5 end
```

Algorithm 7: Perform a cricular queue

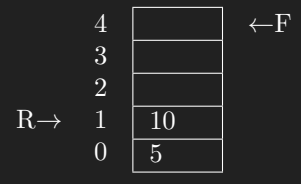
4.8 Circular queue operations

- Let's suppose that we start at an underflow condition.
- REAR = SIZE - 1
- FRONT = SIZE - 1
- SIZE = 5

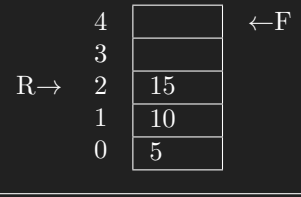

```
CircularQueue.enqueue(5);
// Increase the rear: r = (4+1)%5 -> 0
```




```
CiruclarQueue.enqueue(10);
// rear: (5+1)%5 = 1
```




```
CircularQueue.enqueue(15);
// rear\alpha (6+1)%5 = 2
```




```
CircularQueue.enqueue(20);
// rear: (7+1)%5 = 3
```

	4		←F
R→	3	20	
	2	15	
	1	10	
	0	5	

- An error will occur because in the next insertion will satisfy the underflow condition (rear == front).

```
CircularQueue.enqueue(25);
// rear: (8+1)%5 = 4
```

- This operation will enter the if statement of the queue underflow.

```
if (rear == front){
    printf("Underflow");
}
```

- The insertion will never happen.

- We need to dequeue the last value at index 4.

```
CircularQueue.dequeue();
// This will set the front back to 0
// f = (f+1)%SIZE: (4+1)%5 = 0
```

	4		
R→	3	20	
	2	15	
	1	10	
	0	5	←F

- Now we can move the rear to the next element without satisfying the underflow condition (front == rear).

- The queue is now at an overflow state.

```
CircularQueue.enqueue(25);
// rear:
```

- You can't enqueue anymore because all are occupied.

R→	4	25	
	3	20	
	2	15	
	1	10	
	0	5	←F

4.9 Algorithms for enqueue and dequeue operations for a circular queue

4.9.1 Enqueue

- Initially: item is a 1-D array to hold the queue elements.

```
1 rear = SIZE - 1;
2 front = SIZE - 1;
3 if (rear + 1) % SIZE == front then
4 |   print "Queue overflow";
5 |   exit ENQUEUE;
6 end
7 rear = (rear + 1) % SIZE;
8 item[rear] = v;
9 END ENQUEUE;
```

Algorithm 8: OPERATION ENQUEUE(v)

4.9.2 Dequeue

```
1 if rear == front then
2 |   print "Queue underflow";
3 |   exit DEQUEUE;
4 end
5 front = (front + 1) % SIZE;
6 v = item[front];
7 return v;
8 END DEQUEUE;
```

Algorithm 9: OPERATION DEQUEUE

4.10 Implementation of Circular Queue

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define SIZE 5

typedef struct {
    int item[SIZE];
    int rear;
    int front;
}Queue;

// function prototypes
int dequeue(Queue *);
void enqueue(Queue *, int);
// functionality functions
void initQueue(Queue *);
void menu();
```

```
18
19 void enqueue(Queue *queue, int v){
20     if ((queue->rear + 1) % SIZE == queue->front){
21         printf("Queue overflow\n");
22         return;
23     }
24     queue->rear = (queue->rear + 1) % SIZE;
25     queue->item[queue->rear] = v;
26 }
27 int dequeue(Queue *queue){
28     if (queue->rear == queue->front){
29         printf("Queue underflow\n");
30         return -9999;
31     }
32     queue->front = (queue->front + 1) % SIZE;
33     queue->item[queue->front] = 0; // to observe the effect lets place deleted places
       ↪  to 0
34     int v = queue->item[queue->front];
35     return v;
36 }
37 void initQueue(Queue *queue){
38     queue->front = queue->rear = SIZE - 1;
39 }
40 void menu(){
41     printf("----Queue operations----\n");
42     printf("1. Enqueue\n");
43     printf("2. Dequeue\n");
44     printf("3. Quit\n");
45     printf("--------------------------\n");
46 }
47 void print_queue(Queue *queue){
48     for (int i = 0; i < SIZE; i++){
49         printf("%d: %d\n",i,queue->item[i]);
50     }
51     printf("front: %d, rear: %d\n",queue->front,queue->rear);
52 }
53
54 int main() {
55     menu();
56     Queue q = {};
57     initQueue(&q);
58     bool quit = false;
59     int value;
60
61     while (!quit){
62         int choice;
63         printf("Input your option: ");
64         scanf("%d",&choice);
65
66         switch (choice) {
67             case 1: printf("Input the value to enqueue: ");
68                     scanf("%d", &value);
69                     enqueue(&q, value);
70                     print_queue(&q);
```

```
                    break;
            case 2: value = dequeue(&q);
                    if (value != -9999){
                        printf("Deleted value: %d\n",value);
                    }
                    print_queue(&q);
                    break;
            case 3: quit = true;
                    print_queue(&q);
                    break;
            default: printf("Invalid choice, valid options are 1-3\n");
                     break;
        }
    }
    return 0;
}
/* Output:
----Queue operations----
1. Enqueue
2. Dequeue
3. Quit
--------------------------
Input your option: 1
Input the value to enqueue: 10
0: 10
1: 0
2: 0
3: 0
4: 0
front: 4, rear: 0
Input your option: 1
Input the value to enqueue: 20
0: 10
1: 20
2: 0
3: 0
4: 0
front: 4, rear: 1
Input your option: 1
Input the value to enqueue: 30
0: 10
1: 20
2: 30
3: 0
4: 0
front: 4, rear: 2
Input your option: 1
Input the value to enqueue: 40
0: 10
1: 20
2: 30
3: 40
4: 0
front: 4, rear: 3
```

```
Input your option: 1
Input the value to enqueue: 50
Queue overflow
0: 10
1: 20
2: 30
3: 40
4: 0
front: 4, rear: 3
Input your option: 2
Deleted value: 0
0: 0
1: 20
2: 30
3: 40
4: 0
front: 0, rear: 3
Input your option: 2
Deleted value: 0
0: 0
1: 0
2: 30
3: 40
4: 0
front: 1, rear: 3
Input your option: 2
Deleted value: 0
0: 0
1: 0
2: 0
3: 40
4: 0
front: 2, rear: 3
Input your option: 1
Input the value to enqueue: 50
0: 0
1: 0
2: 0
3: 40
4: 50
front: 2, rear: 4
Input your option: 1
Input the value to enqueue: 60
0: 60
1: 0
2: 0
3: 40
4: 50
front: 2, rear: 0
Input your option: 1
Input the value to enqueue: 70
0: 60
1: 70
2: 0
```

```
3: 40
4: 50
front: 2, rear: 1
Input your option: 1
Input the value to enqueue: 80
Queue overflow
0: 60
1: 70
2: 0
3: 40
4: 50
front: 2, rear: 1
Input your option: 3
0: 60
1: 70
2: 0
3: 40
4: 50
front: 2, rear: 1

*/
```

4.11 Introduction to Double Ended Queue

- In a double ended queue, you can insert or delete front either side of the queue, it's a mixture of the stack and FIFO queue data structure.

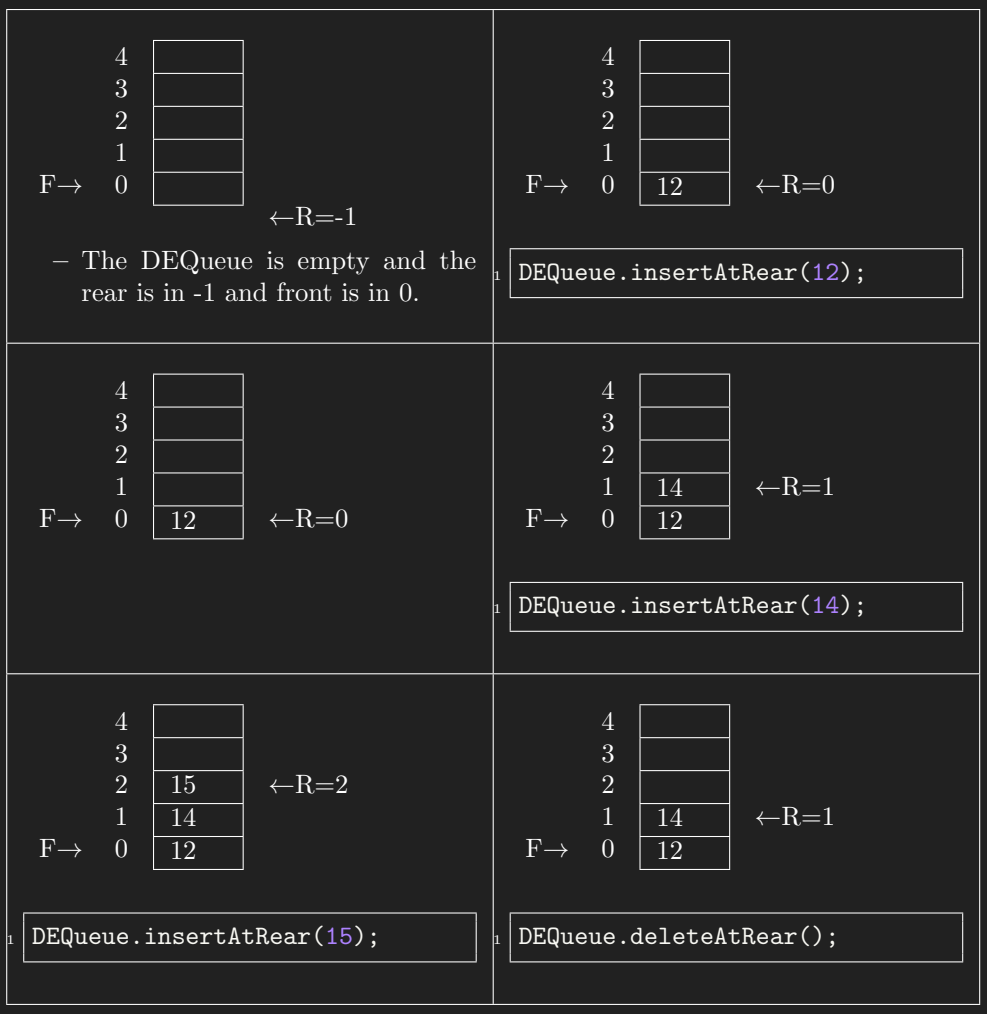
- The operations are:

  - Inserting at REAR.
  - Deletion from REAR.
  - Inserting at FRONT.
  - Deleting from FRONT.

4.11.1 Example

- Insertion at REAR.

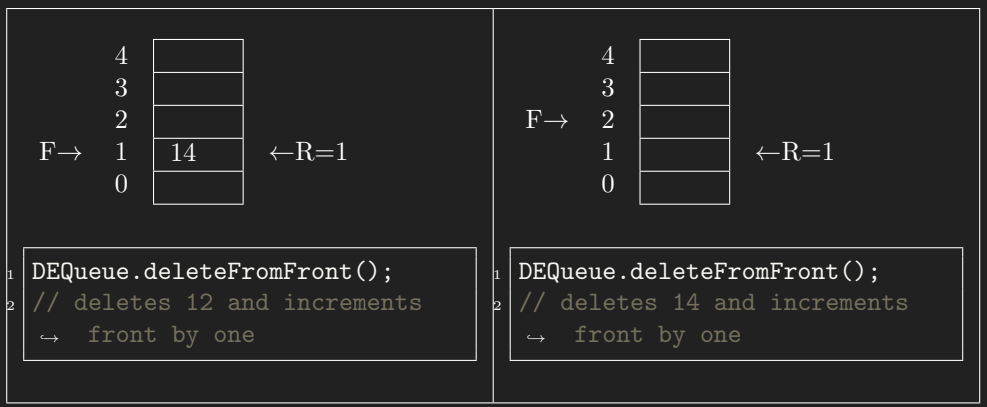
  - Inserting at rear and deleting from rear are exactly the same as the push and pop operations we had in the stack.
  - Assume:
    - REAR = -1
    - FRONT = 0



- The overflow condition for the insert at REAR is going to be when R=4.

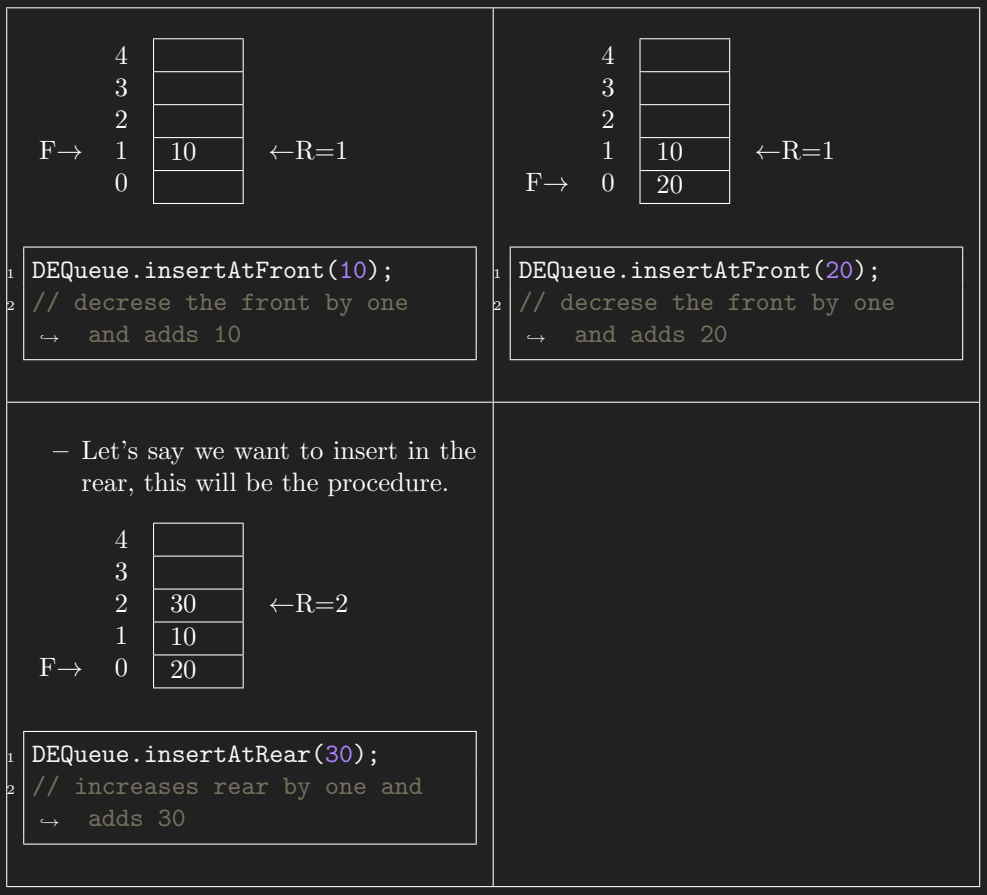
- Insertion at FRONT: when we have inserted with rear elements, and we have a situation in which the front is at index 0 and the rear is in index 2, when the rear is higher in terms of index than the front the insertion at FRONT will not be possible, this is the overflow condition.

- Perform deletions in a double ended queue, using the deletion from the FRONT.



- Note that the queue is empty, we can not perform any more deletion.
- The underflow state happens when the front is greater than the rear.

- Now lets try an insertion from the front.



4.12 Algorithm development for double ended queue operations

Take the following considerations:

- FRONT = 0
- REAR = -1
- SIZE = 5

4.12.1 Insertion at rear

```
1 if rear == SIZE - 1 then
2 |   print "Unable to add at rear";
3 |   exit INSERTION_OPERATION;
4 end
5 rear = rear + 1;
6 item[rear] = v;
7 END INSERTION AT REAR;
```

Algorithm 10: OPERATION INSERTION AT REAR (v)

4.12.2 Delete from rear

```
1 if front > rear then
2 |   print "Queue underflow";
3 |   exit DELETE FROM REAR;
4 end
5 v = item[rear];
6 rear = rear - 1;
7 return v;
8 END OF DELETE FROM REAR;
```

Algorithm 11: OPERATION DELETE FROM REAR

4.12.3 Insert at front

```
1 if front == 0 then
2 |   print "Unable to insert at front";
3 |   exit OPERATION INSERT AT FRONT;
4 end
5 front = front - 1;
6 item[front] = v;
```

Algorithm 12: OPERATION INSERT AT FRONT

4.12.4 Deletion from front

```
1 if front > rear then
2 |   print "Queue underflow";
3 |   exit DELETE FROM FRONT;
4 end
5 v = item[front];
6 front = front + 1;
7 return v;
8 END DELETION FROM FRONT;
```

Algorithm 13: OPERATION DELETION FROM FRONT

4.13 Implementation of double ended queue

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define SIZE 5

typedef struct {
    int item[SIZE];
    int rear;
    int front;
}Queue;

```

```c
// function prototypes
void insertAtRear(Queue *, int);
int deleteFromRear(Queue *);
void insertAtFront(Queue *, int);
int deleteFromFront(Queue *);
void initQueue(Queue *);

void insertAtRear(Queue *queue, int v){
    if (queue->rear == SIZE - 1){
        printf("Unable to add at rear\n");
        return;
    }
    queue->rear++;
    queue->item[queue->rear] = v;
}
int deleteFromRear(Queue *queue){
    if (queue->front == queue->rear){
        printf("Queue underflow\n");
        return -9999;
    }
    int v = queue->item[queue->rear];
    queue->item[queue->rear] = 0;
    queue->rear--;
    return v;
}
void insertAtFront(Queue *queue, int v){
    if (queue->front == 0){
        printf("Unable to insert at front\n");
        return;
    }
    queue->front++;
    queue->item[queue->front] = v;
}
int deleteFromFront(Queue *queue){
    if (queue->front > queue->rear) {
        printf("Queue underflow\n");
        return -9999;
    }
    int v = queue->item[queue->front];
    queue->item[queue->front] = 0;
    queue->front++;
    return v;
}
void menu(){
    printf("----Queue operations----\n");
    printf("1. Insert at rear\n");
    printf("2. Insert at front\n");
    printf("3. Delete from rear\n");
    printf("4. Delete from front\n");
    printf("-------------------------\n");
}
void print_queue(Queue *queue){
    for (int i = 0; i < SIZE; i++){
        printf("%d: %d\n",i,queue->item[i]);
```

```
    }
    printf("front: %d, rear: %d\n",queue->front,queue->rear);
}
void initQueue(Queue *queue){
    queue->front = 0;
    queue->rear = -1;
}

int main() {
    menu();
    Queue q = {};
    initQueue(&q);
    bool quit = false;
    int value;

    while (!quit){
        int choice;
        printf("Input your option: ");
        scanf("%d",&choice);

        switch (choice) {
            case 1: printf("Input the value for rear insertion: ");
                    scanf("%d", &value);
                    insertAtRear(&q,value);
                    print_queue(&q);
                    break;
            case 2: printf("Input the value for front insertion: ");
                    scanf("%d", &value);
                    insertAtFront(&q,value);
                    print_queue(&q);
                    break;
            case 3: printf("Deleted item from rear: %d\n",deleteFromRear(&q));
                    print_queue(&q);
                    break;
            case 4: printf("Deleted item from front: %d\n",deleteFromFront(&q));
                    print_queue(&q);
                    break;
            default: printf("Invalid choice, valid options are 1-4\n");
                     break;
        }
    }
    return 0;
}
/* Output:
----Queue operations----
1. Insert at rear
2. Insert at front
3. Delete from rear
4. Delete from front
------------------------
Input your option: 1
Input the value for rear insertion: 10
0: 10
1: 0
```

```
2: 0
3: 0
4: 0
front: 0, rear: 0
Input your option: 1
Input the value for rear insertion: 15
0: 10
1: 15
2: 0
3: 0
4: 0
front: 0, rear: 1
Input your option: 1
Input the value for rear insertion: 20
0: 10
1: 15
2: 20
3: 0
4: 0
front: 0, rear: 2
Input your option: 2
Input the value for front insertion: 7
Unable to insert at front
0: 10
1: 15
2: 20
3: 0
4: 0
front: 0, rear: 2
Input your option: 4
Deleted item from front: 10
0: 0
1: 15
2: 20
3: 0
4: 0
front: 1, rear: 2
*/
```

Chapter 5

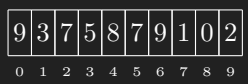
Linked List

5.1 Introduction to linked lists

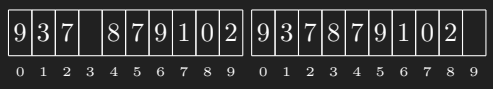
- Linked lists resolve the problems commonly encountered with arrays, that is the fact that arrays have to be declared of a fixed size and can't really be dynamically grown without fragmenting memory is a problem we can resolve using a linked list data structure.
- The draw backs of arrays are actually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

5.1.1 What is wrong with arrays?

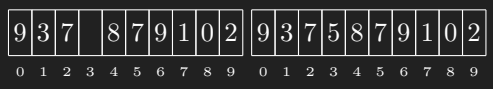
- The first issue with array is the size of the array needs to be declared. Using realloc() will not really resolve this because it will copy all the elements of the array in to a bigger memory location, this causes poor performance and fragmentation.
- Once allocated with a size it is a hazard to change 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.

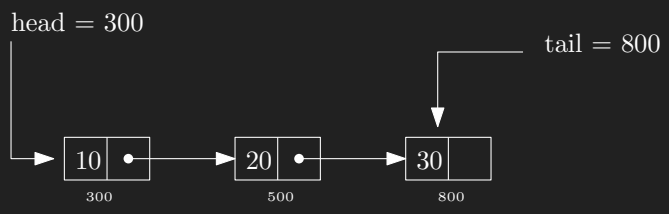
5.2 Definition of linked list, conception of node, understanding basic principles

- 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.

5.2.1 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



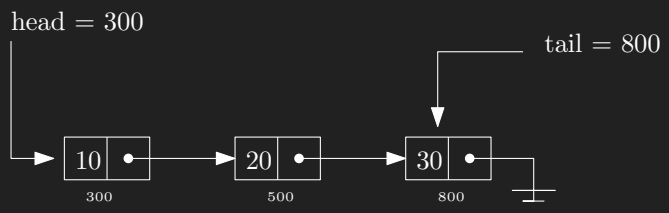
Every node stores pointers to the next or sometimes to the previous nodes.

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

5.3.1 Singly linked lists

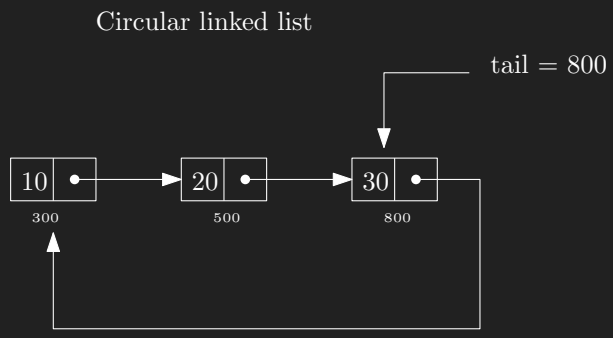
- Consists of nodes containing a data and pointer member, each pointer points to the next node and the tail is recognized when a null pointer is encountered.

Single linked list



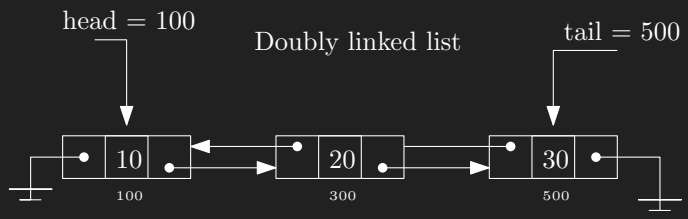
5.3.2 Circular linked list

- 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.



5.3.3 Double linked list

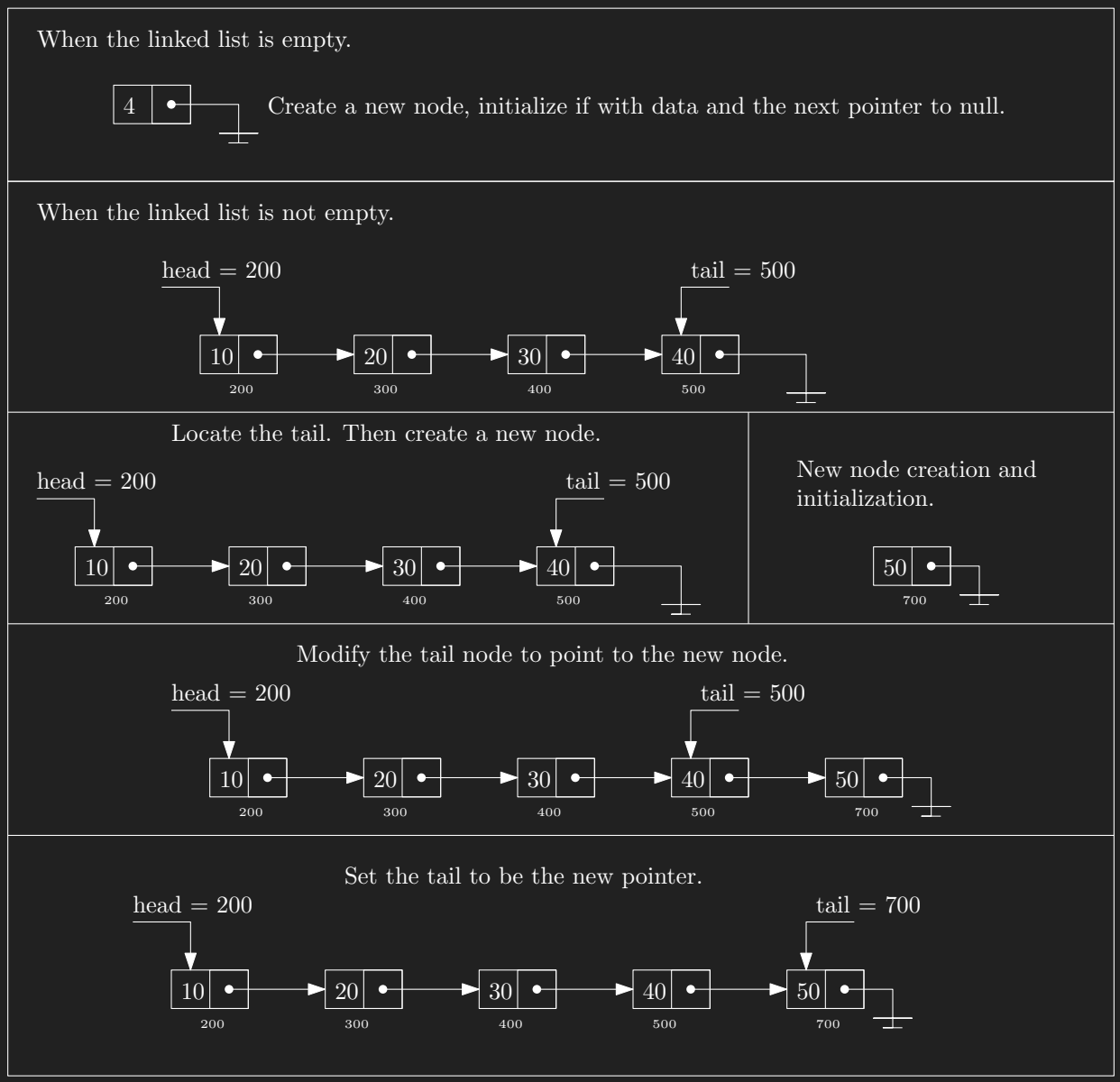
- In this linked list, each node has two pointers, one that points to the next node and the other that points to the previous node.
- The previous pointer in the first node and the next pointer in the terminal node are always pointing to null.
- This is a way of implementing a dynamic double ended queue.



Chapter 6

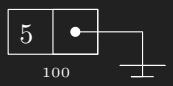
Singly linked lists

6.1 Linked list operation for insert at tail



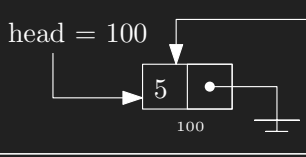
6.2 Linked list operation for inserting at head

If the linked list is empty.

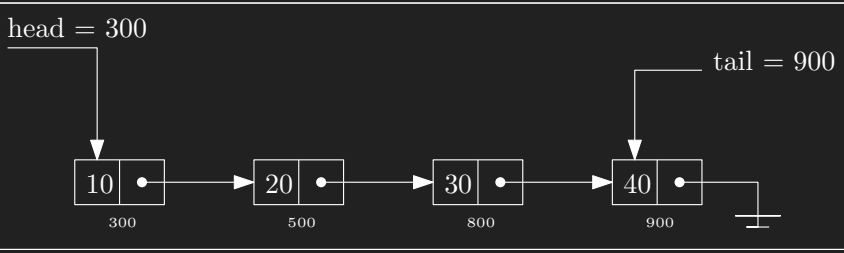


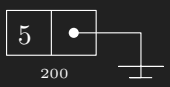
Create a new node and initialize it with data and set the next pointer to null.

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



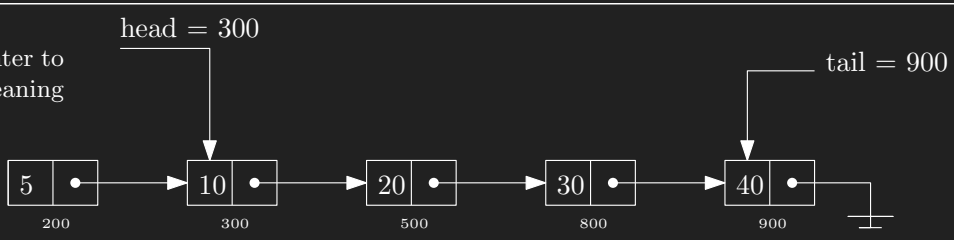
If the linked list is not empty.

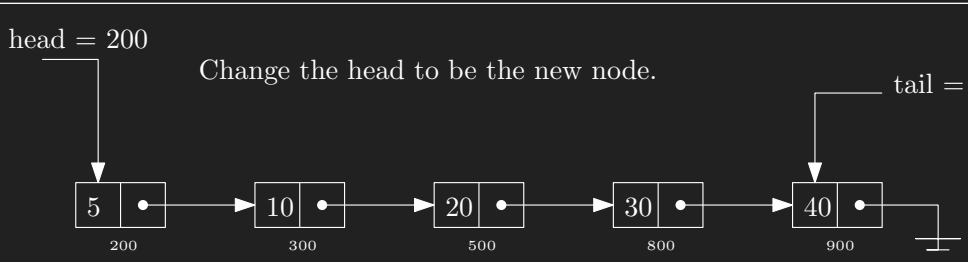




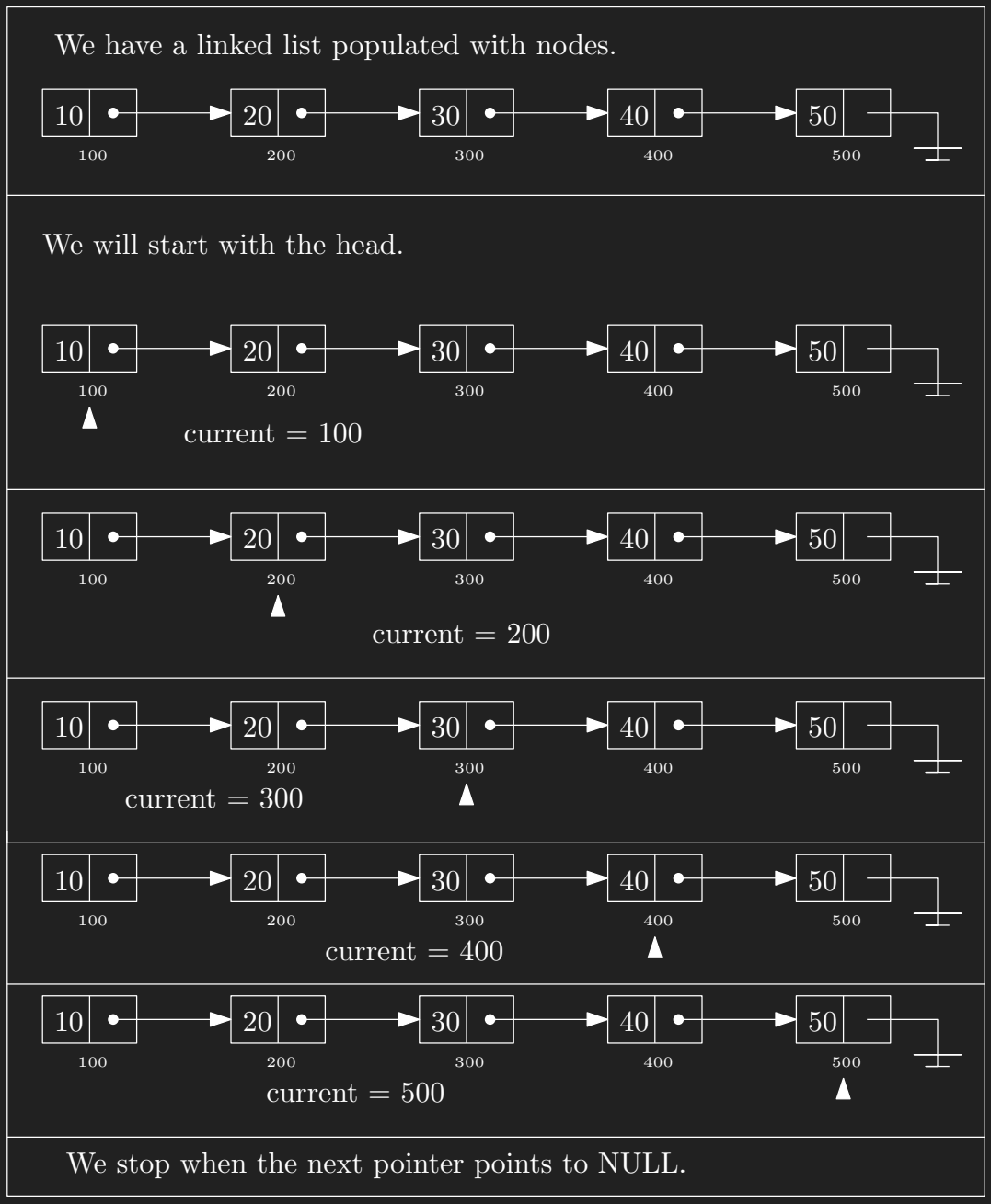
Create a new node object in memory and set the next pointer to NULL and store the data.

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

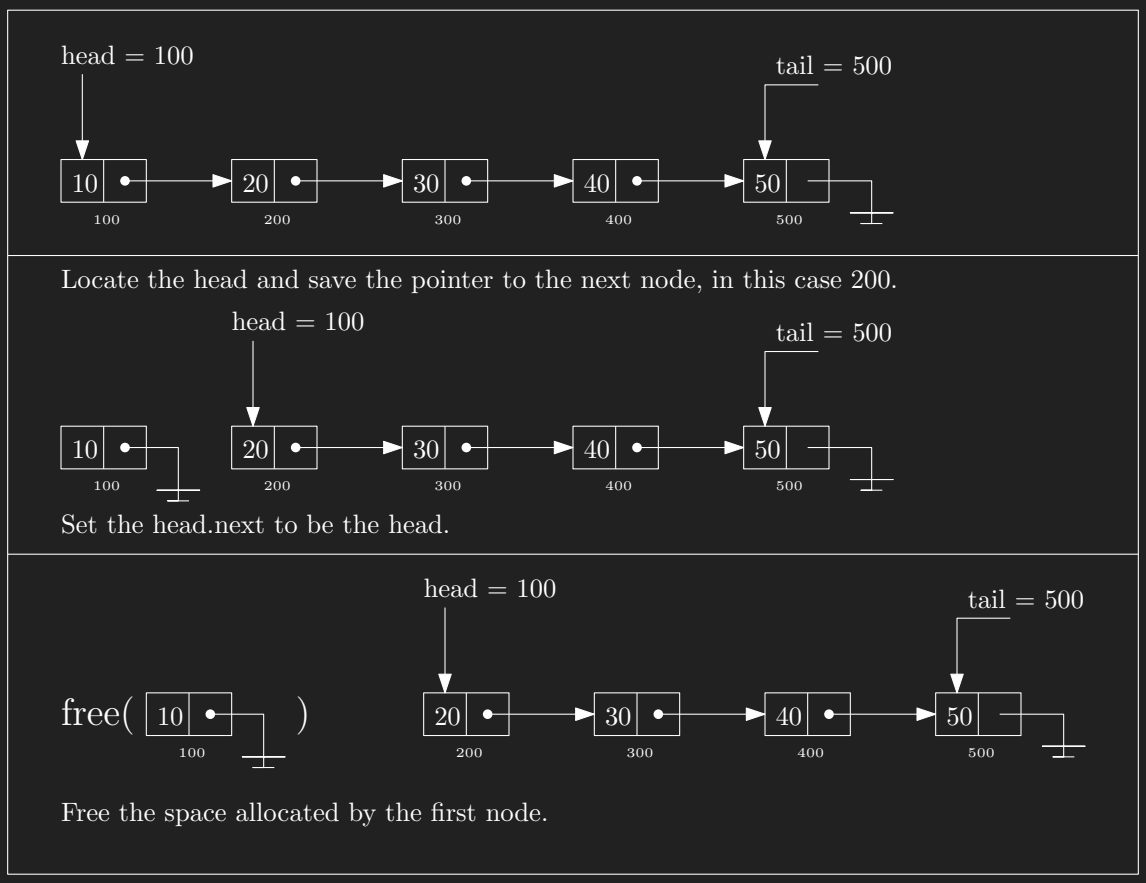




6.3 Linked list operation for traversing singly linked lists



6.4 Linked list operation for delete first



6.5 Linked list operation for delete last

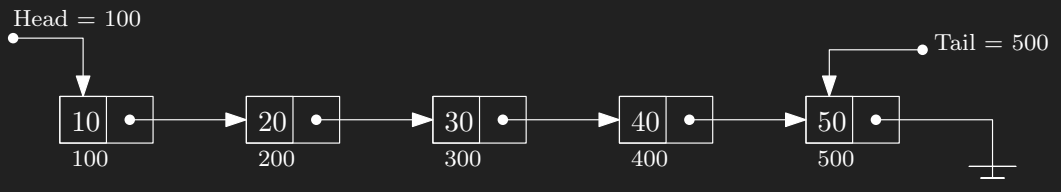
6.6 Linked list operation for deleting target nodes

6.7 Linked list operation for finding a node

6.8 Linked list operation reverse a singly linked list

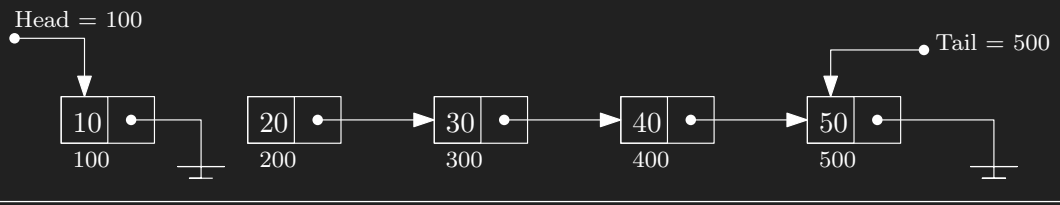
Reversing algorithm for singly linked list

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



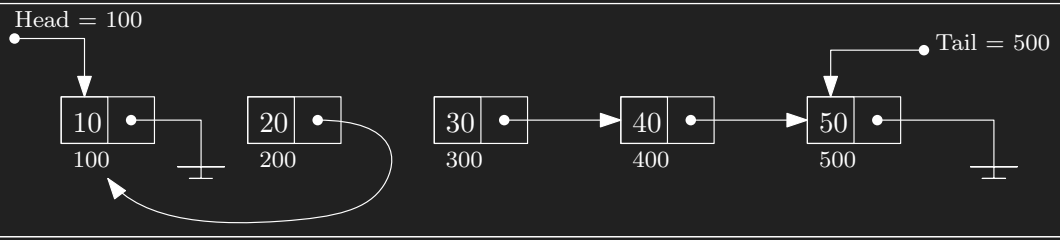
1

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



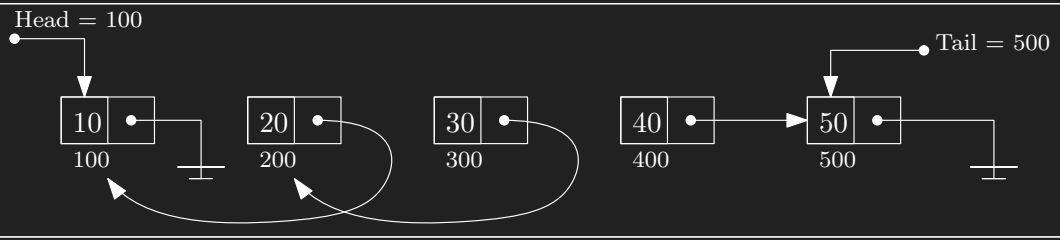
2

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



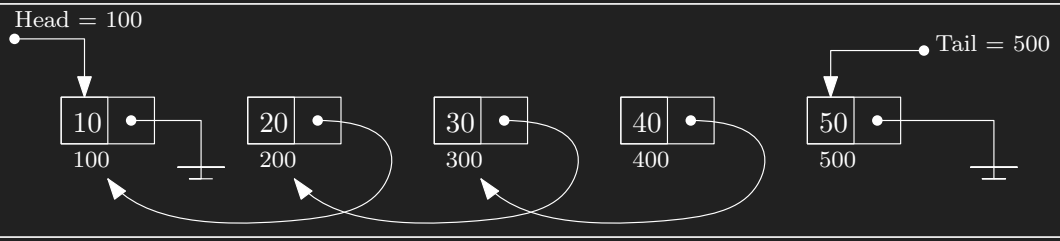
3

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



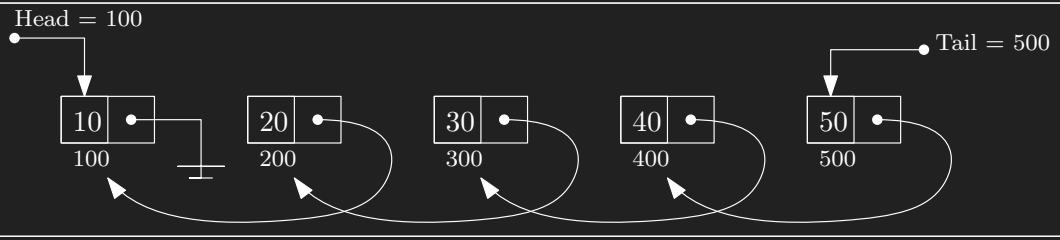
4

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



5

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



6

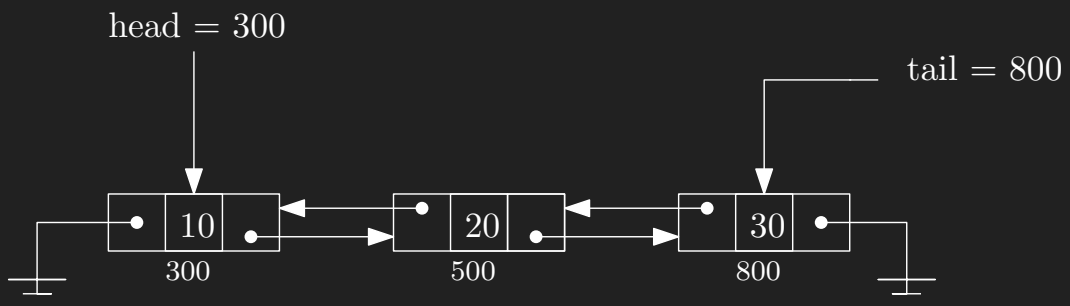
head = 500;
tail = 100;

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

Chapter 7

Doubly linked list

7.1 Introduction to doubly linked list



7.2 Doubly linked list operation for adding node at head

7.3 Doubly linked list operation for adding node at tail

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

7.5 Doubly linked list operation for deleting node at head

7.6 Doubly linked list operation for deleting node at tail

7.7 Doubly linked list operation for deleting a target node

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

Chapter 8

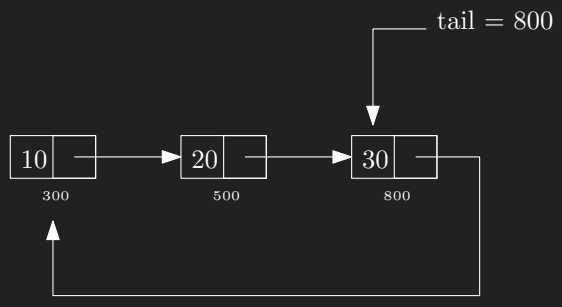
Circular linked lists

8.1 Introduction

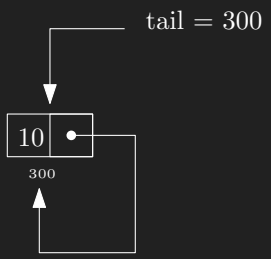
- In a circular linked lists we have pretty much the same situation as a singly linked list, the difference is that we have only one pointer to the list, the tail, and the tail.next does not point to NULL, instead it points to the first element of the list.
- We only keep track of the tail, considering 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.

8.1.1 Circular linked list visualization

A populated circular linked list.



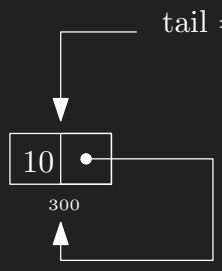
When there is only one node, the list looks like this.



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

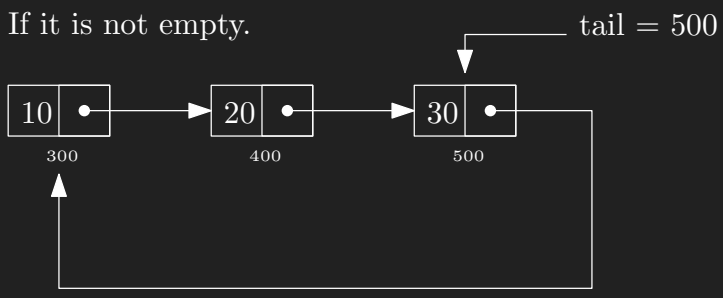
8.2 Operation for inserting a node to circular linked list

If the list is empty.

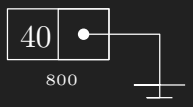


Create a new node in memory, then initialize it with data, finaly make the .next pointer point to itself, this case: data = 10, .next = 300.

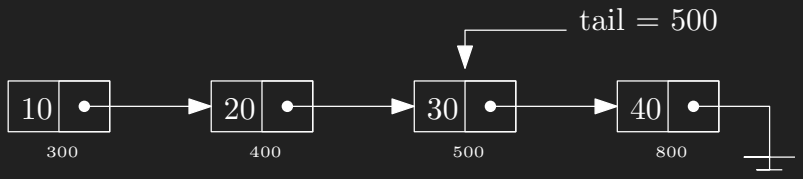
If it is not empty.



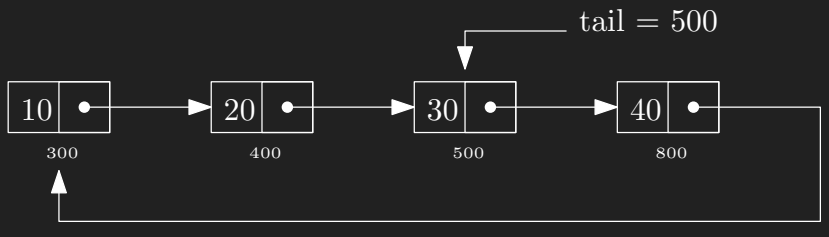
First, create a new node instance and initialize it.



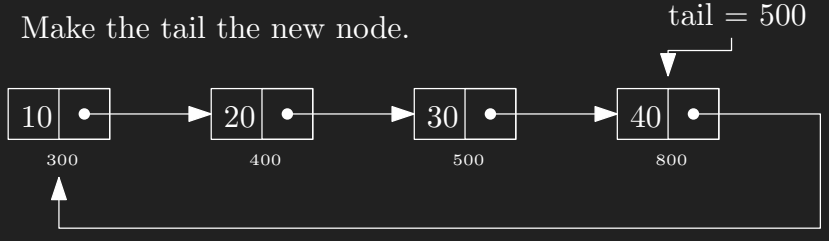
Make tail.next point to the new node.



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



Make the tail the new node.



Chapter 9

Efficiency of an algorithm

9.1 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.

9.1.1 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.

9.1.2 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.

9.1.3 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 $n > 0$.

9.2 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.

9.2.1 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.

9.3 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 $T \leq c \times n^2$

9.3.1 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>;
}
```

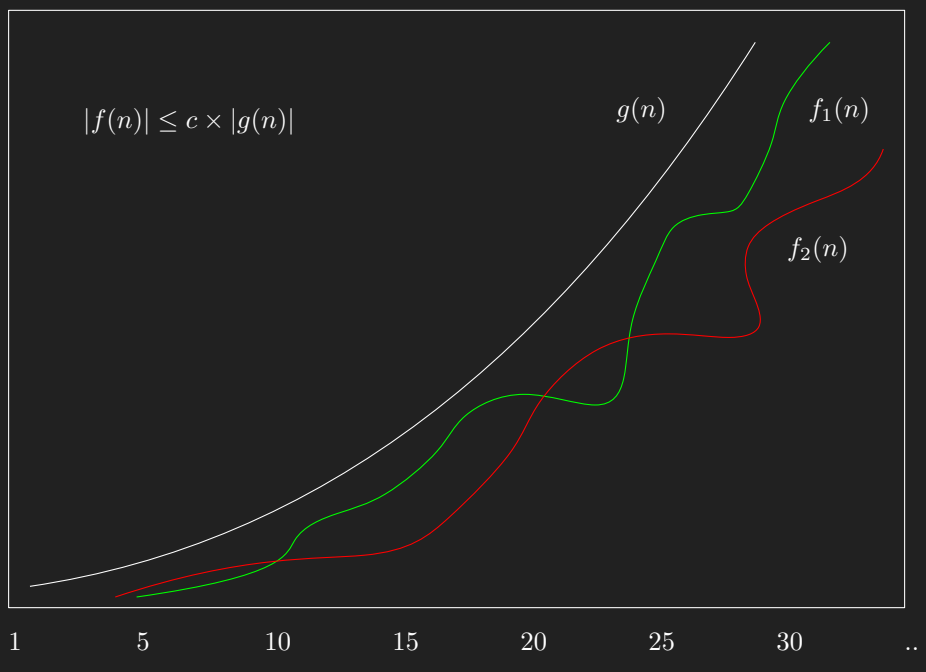
- Defining c as a positive integer constant.
- For any algorithm, if you find that the complexity is something like $f(n) \leq c \times n^2$ we say that the complexity $O(n^2)$ (Big-O of n^2) this is the worse case scenario.
- For any algorithm we find that has a complexity of $f(n) \leq c \times n \log(n)$ 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:

$$O(g(n)) \quad \Longrightarrow \quad f(n) \leq c \times g(n)$$

  - 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.

9.3.2 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 $f(n) \in O(f(n))$



9.4 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:

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

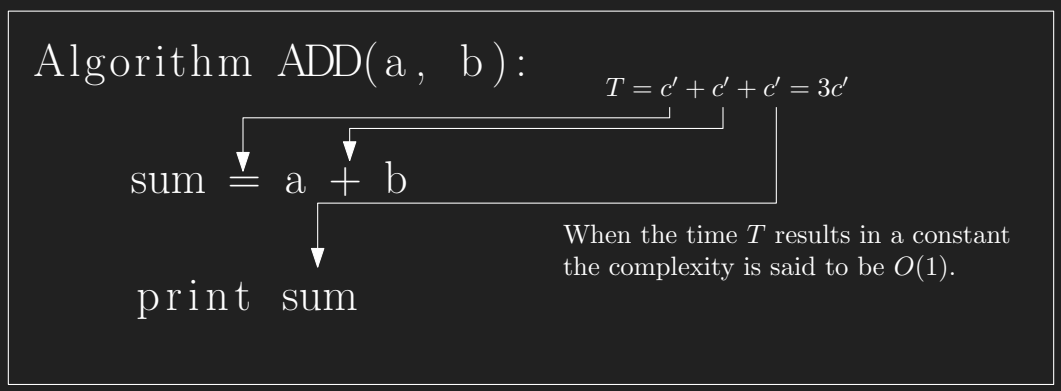
Another example:

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

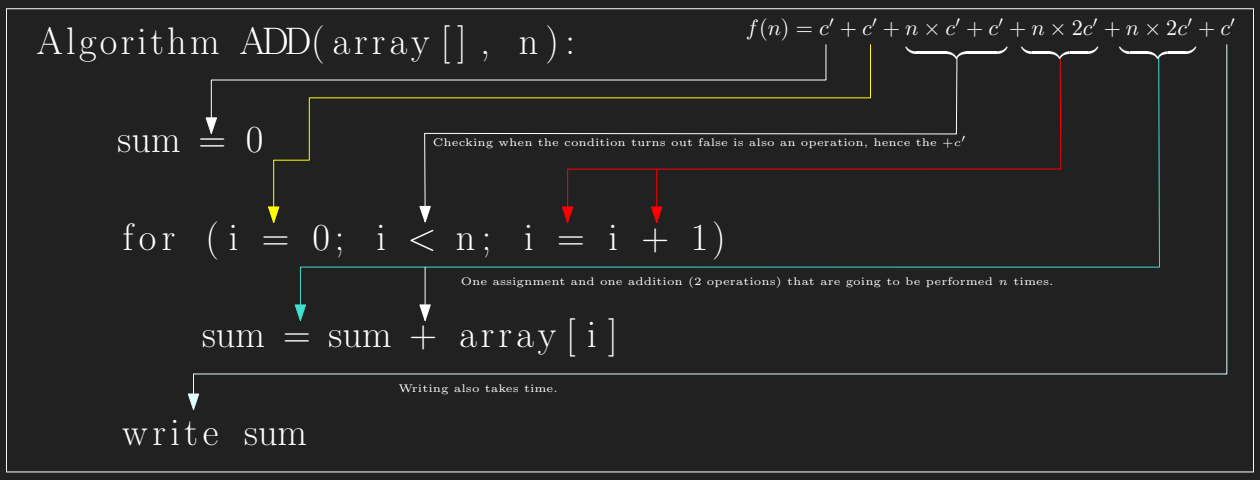
9.4.1 Finding Big-O

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.



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.



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

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 $O(n)$.

9.5 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:

$$f(n) = c + f(n-1)$$
$$f(n-1) = c + f(n-2)$$

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

$$f(n) = c + (c + f(n-2))$$
$$f(n) = 2c + f(n-2)$$
$$f(n-2) = c + f(n-3)$$

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

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

9.5.1 Another example

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

9.6 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.

$$\frac{n}{2^k} = 1$$
$$n = 2^k$$
$$k = \log_2(n)$$

Thus the total time taken for this algorithm is:

$$T = c \times \log_2(n) + c$$
$$T = c' \log_2(n)$$

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

9.6.1 The same but evaluated with recurrence

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

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

- 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:

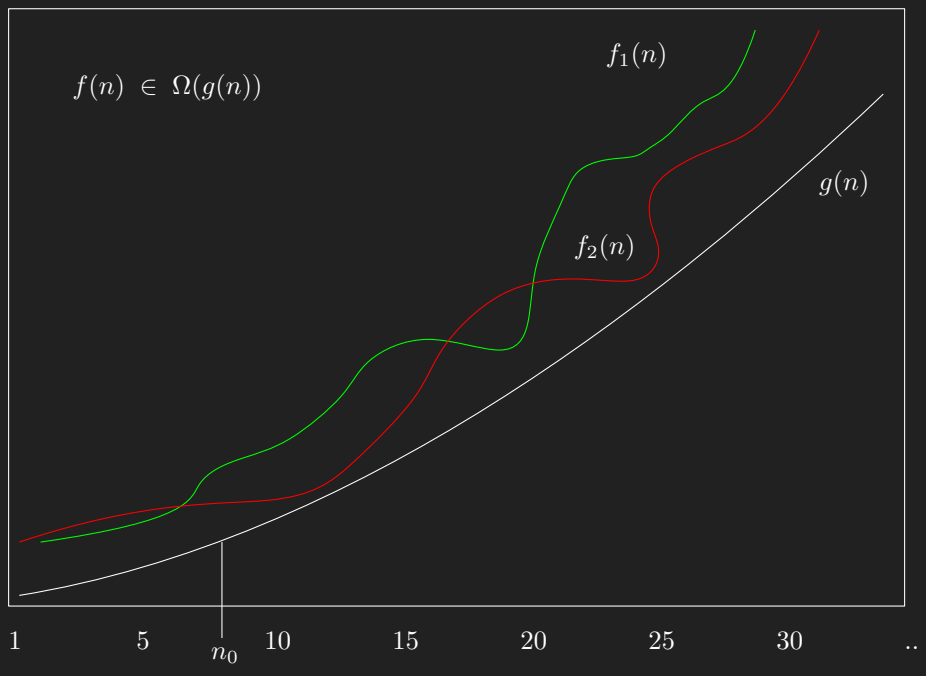
$$f(n) \leq c' \times \log_2(n)$$
$$\therefore O(\log(n))$$

9.7 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.



9.7.1 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.

9.7.2 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.

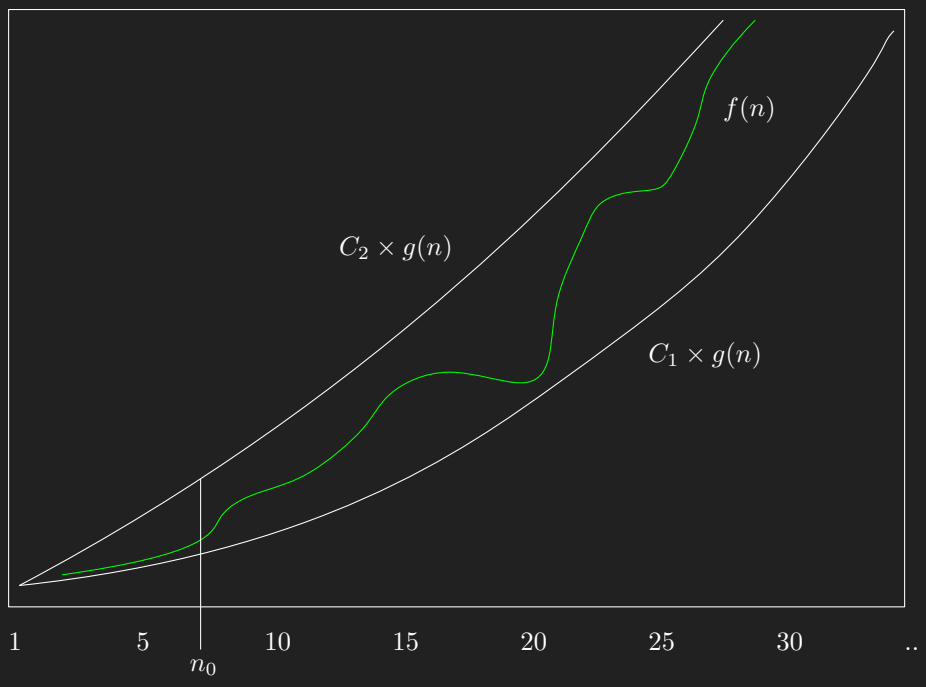
On the other case, if in the very first iteration k is inputted to be odd, this will break the outer loop condition and make it so that only the inner loop executes n times, this gives us the best case complexity of $\Omega(n)$.

9.8 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).



9.8.1 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 10

Binary search

10.1 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.

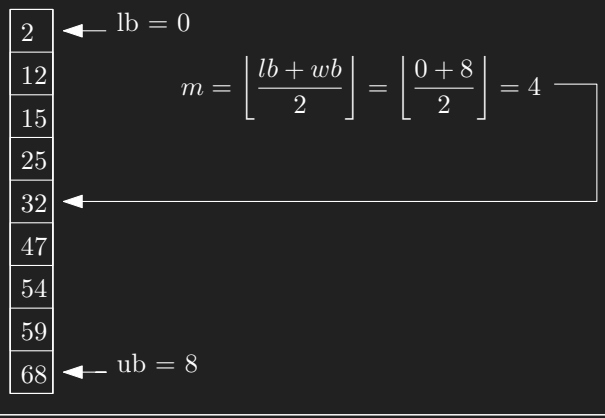
- The list or array must be in order, ascending or descending, but in order.

10.1.1 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.

10.1.2 Searching the upper part

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

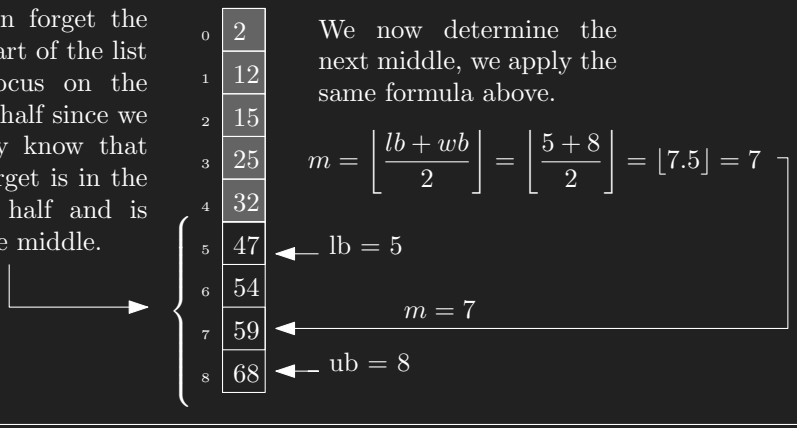


Compare the middle element and check whether it contains the target (59). If it doesn't contain the target compare if the target is bigger or smaller than the middle element; if the target is greater the middle might be in the upper half, if the target is less than the middle then the target might be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



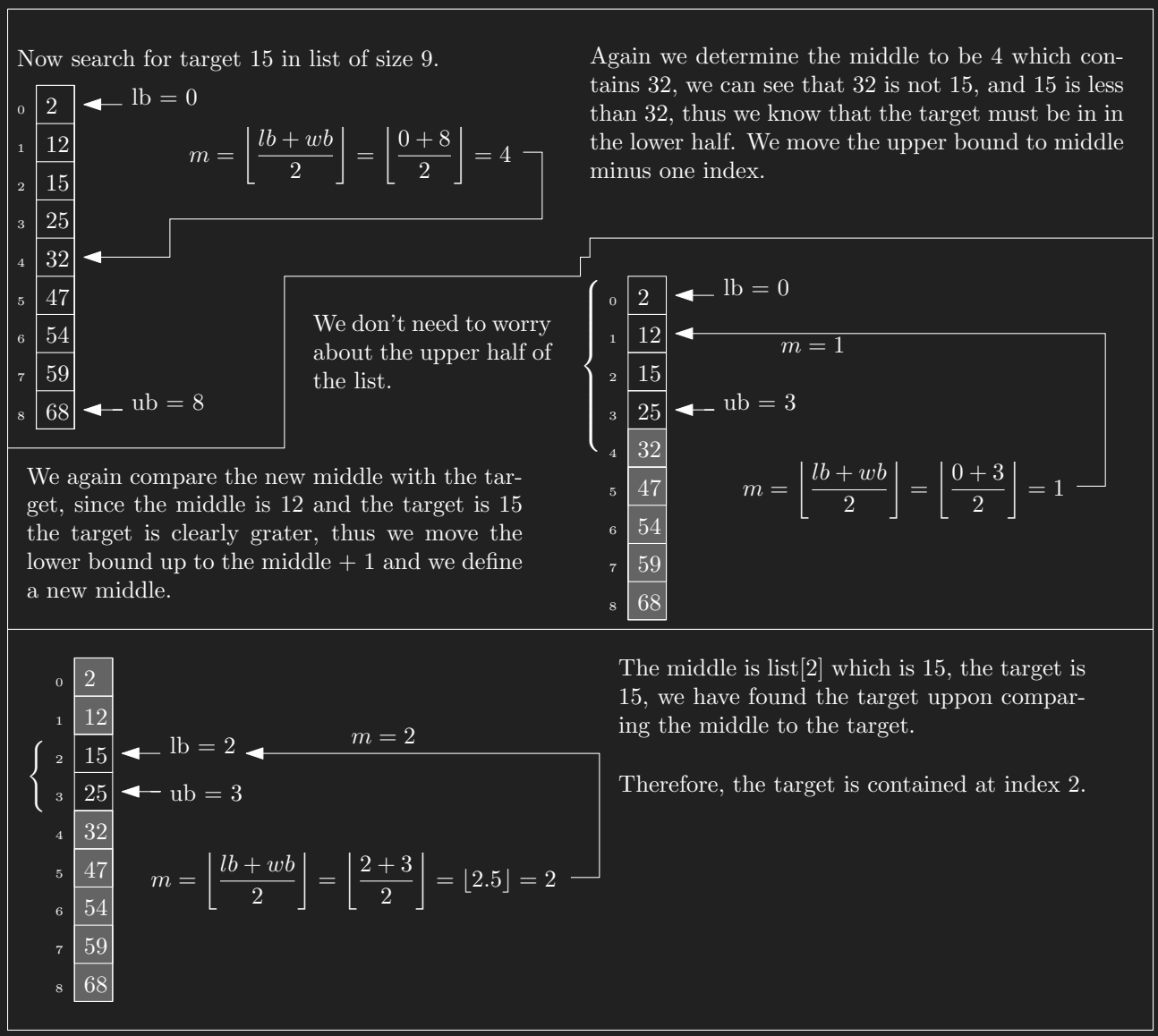
We now determine the next middle, we apply the same formula above.

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

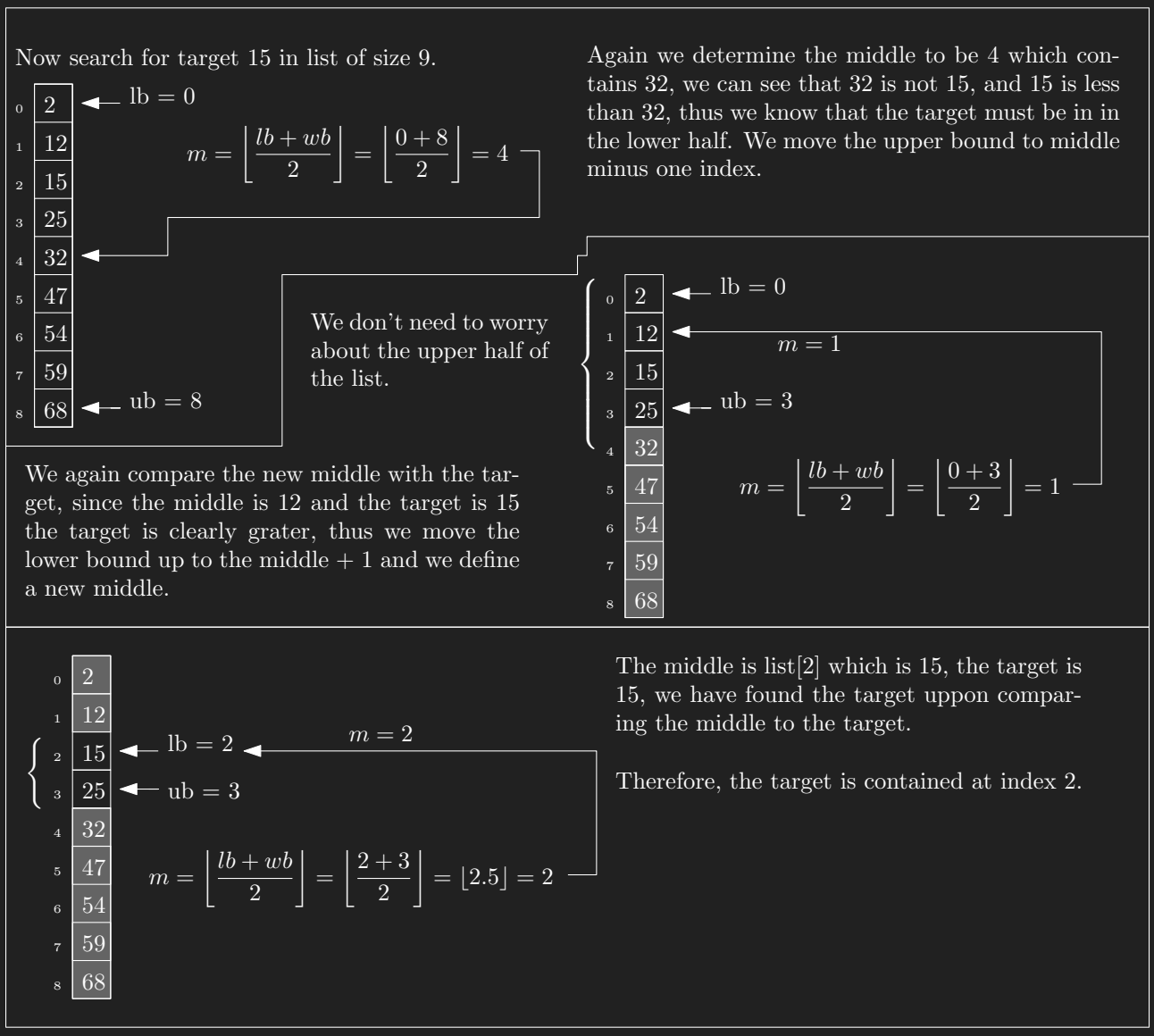
Now we compare if the middle is the target, it indeed is, we now have found the target in the list with as little operations as posible.

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

10.1.3 Searching the lower part



10.1.4 When the target is not on the list



10.2 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
*/
```

10.3 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 11

Recursion

11.1 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.

11.2 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.

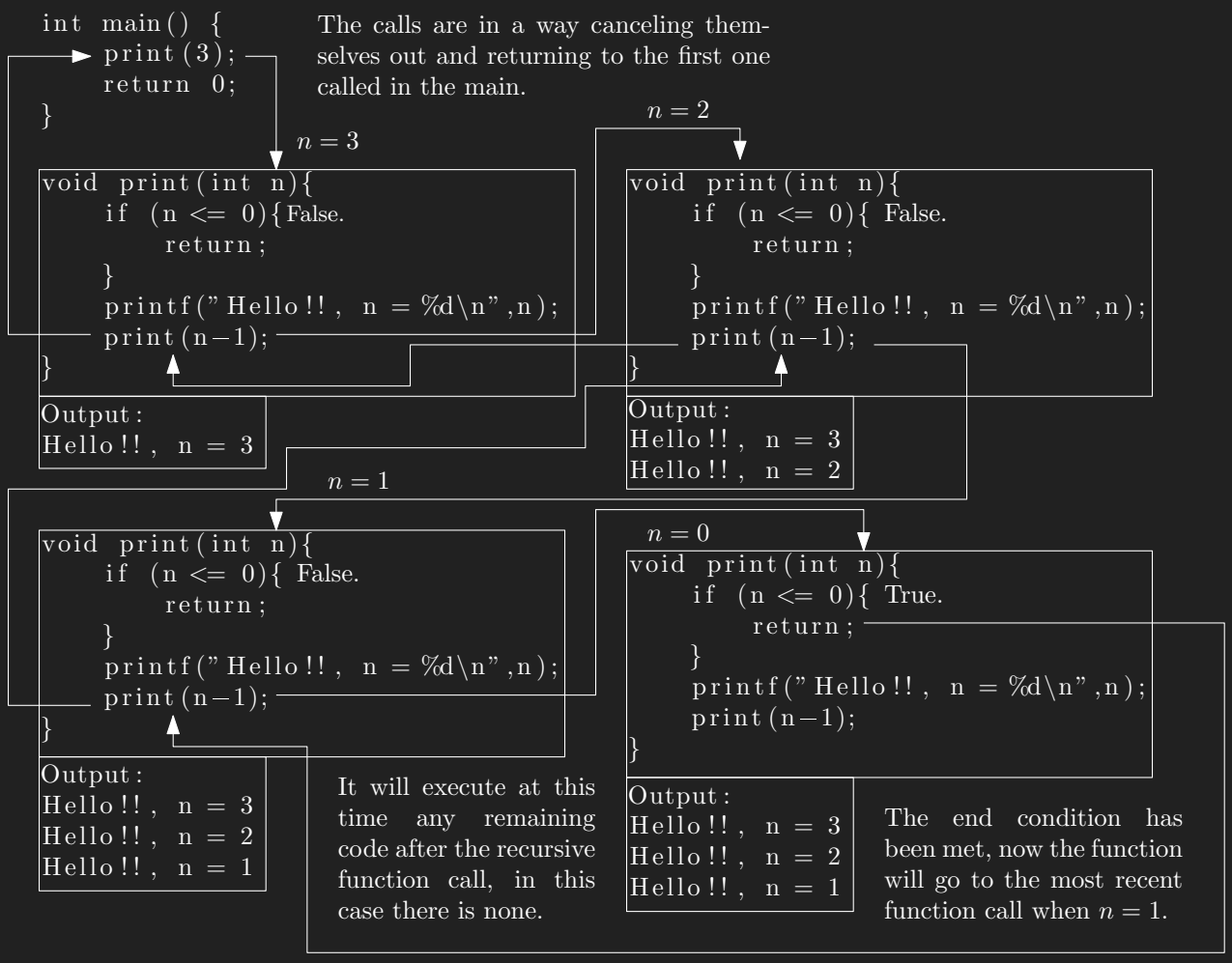
11.3 When and how to terminate — the base condition of recursion

- 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

11.4 Let us go into the depth of the call



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

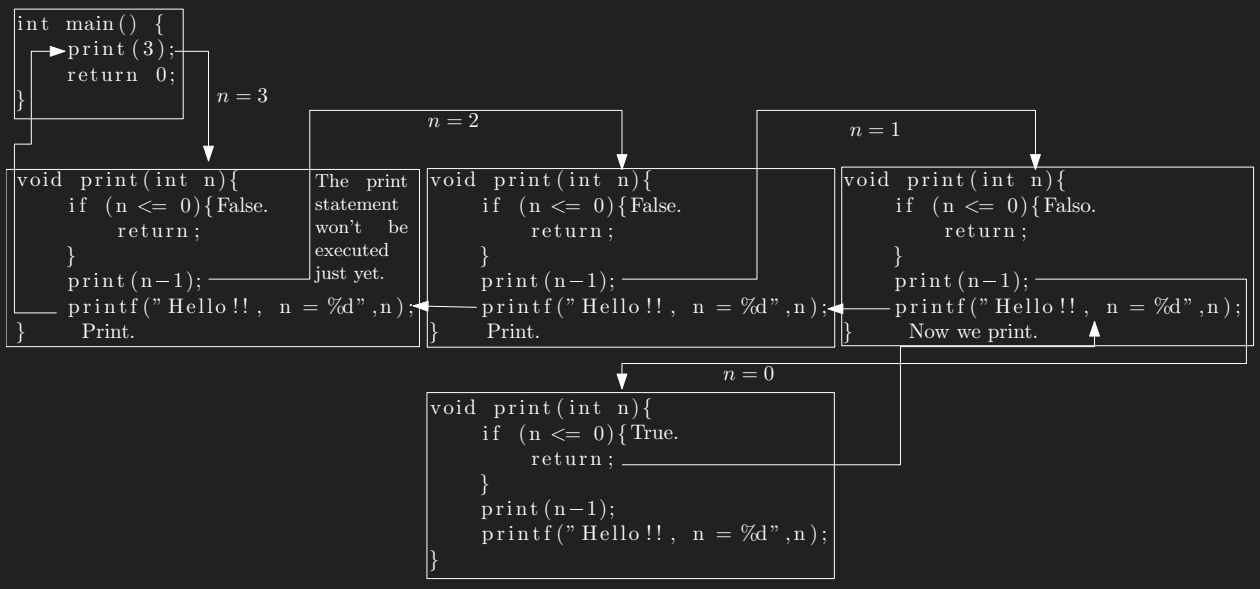
```
    void print(int n){
    if (n <= 0){
        return;
    }
    printf("Hello!!\n");
    print(n-1); // recursive call
}
/* Output:
Hello!!, n = 3
Hello!!, n = 2
Hello!!, n = 1

*/
```

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

```
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

*/
```



11.5 Recursion example — Juggler Sequence

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

```c
#include <stdio.h>
#include <stdlib.h>
#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,
*/
```

11.6 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;
}
```

11.7 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
*/
```

11.8 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

*/
```

11.9 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.

11.10 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.

11.11 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.

11.12 Recursion versus iteration

This is a debate and every programmer must know this.

11.12.1 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.

```
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.
}
```

11.12.2 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.

11.12.3 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.

11.12.4 Synthesis in a programmer's way

```
if you can convert the recursion to tail recursion then
    Iteration and recursion are equivalent;
    # 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.
if we can not convert the loop to tail recursion && it doesn't require a stack then
    iteration is better;
if when we need a stack explicitly to solve a problem then
    recursion is better;
    # This is preferred rather to implementing our own stack.
```

Algorithm 14: When to use recursion or iteration

Chapter 12

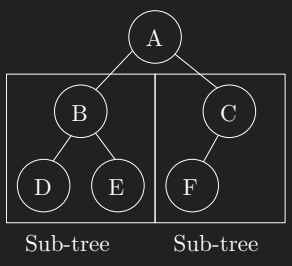
Binary tree and binary search tree

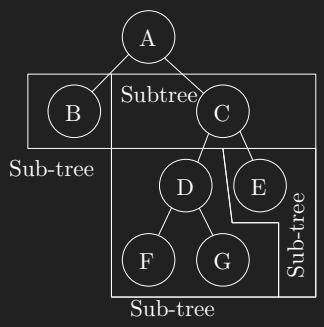
12.1 Introduction to binary tree

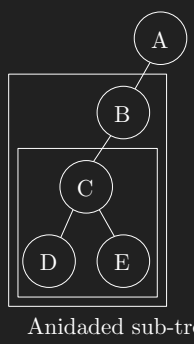
- In a binary search tree, the elements have a hierarchical relationship or *parent-child* relationship. These are not linear data structures, linked lists, arrays, stacks, queues are all linear because they do not have a hierarchy.
- Any tree has hierarchical relationships.

12.1.1 Examples of binary trees

- A tree can have 0,1 or at most two children.
- Each child has 0 or 1 parent.
- The root node is the only node that can have no parents.
- A root can have an empty right tree and an empty left tree.
- The definition of binary trees is recursive, since for instance we can remove node A and be left with two trees by themselves, in this case B would be the root for the left tree, and C would be the root for the right tree.

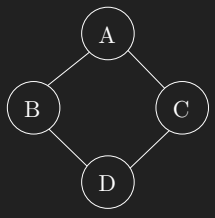






Anidaded sub-tree.

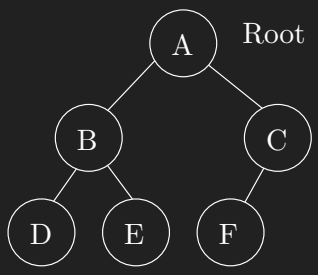
A sub-tree in itself.



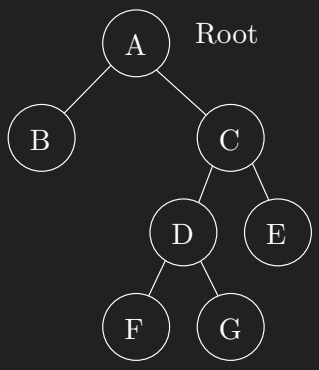
Node D has two parents. We can also see it's not a tree because there is no hierarchy.

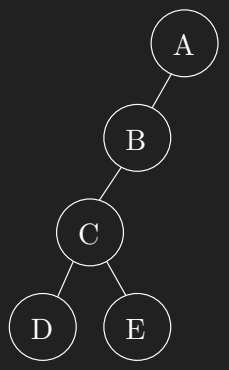
12.1.2 Valid examples and non-valid examples

Examples of valid binary trees



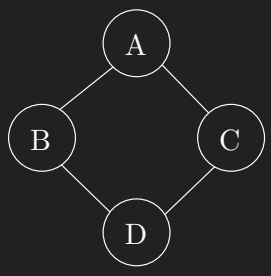
In all examples node A is the root.





We can have a root by itself and this can be considered a tree.

Examples of non binary trees



Node D has two parents. Has no hierarchy. This is actually a graph.

12.2 Formal definition

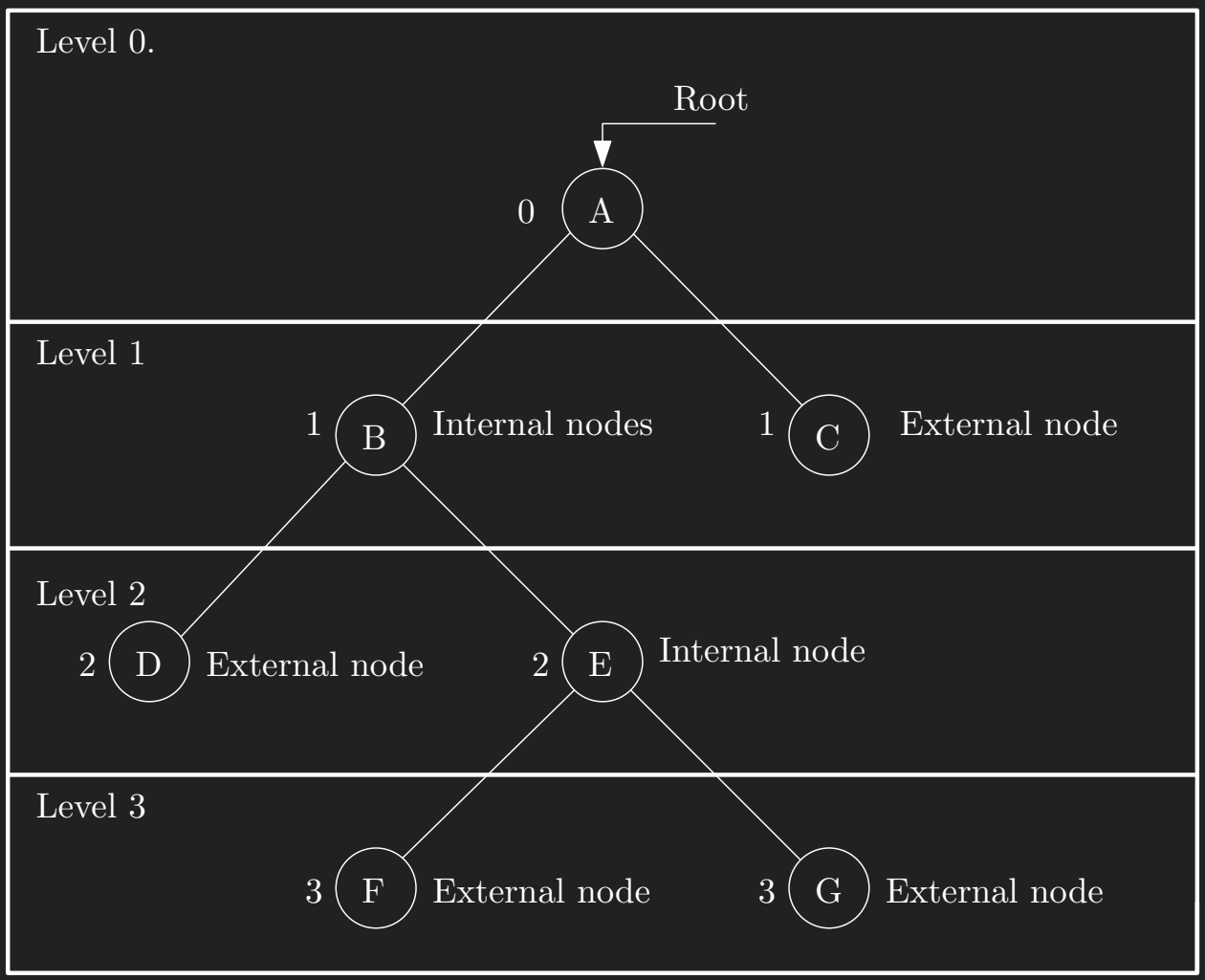
Definición de "Binary tree": *Binary tree is a set of 3* disjoint *subsets, the first one is the root of the tree and the other 2 subsets are either empty or they are themselves binary trees.[***DEusingCCpp***]*

12.3 Understanding different terminologies related with binary trees

- The root is located at level 0.
- The children of any parent node will have a level equivalent to the level of the parent plus one.
- A leaf node or external node is a node that does not have any children.
- An internal node is a parent node with two children.
- A half-leaf node is a parent node with only one child.
- The height of the binary tree is the level of the leaf that is at the bottom, some books have it as the level of the leafs at the bottom + one.
- Two nodes are considered siblings when they are children of the same parents.

12.3.1 Example

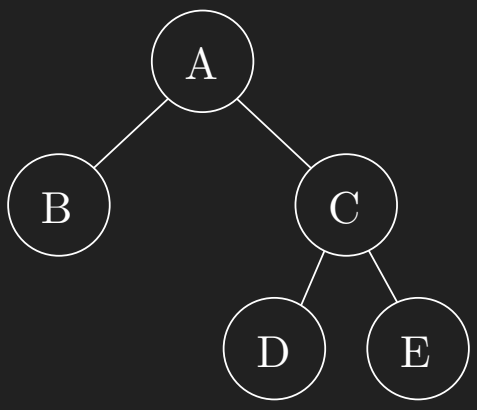
- Root is A.
- C, D, F & G are leaf nodes or external nodes.
- A, B & E are internal nodes.
- There are no half-leafs except for node A.
- The height is 3 (considering start at 0).
- $(B, C), (D, E)$ & (F, G) are siblings because they have the same parent.



12.4 Two tree / strictly binary tree

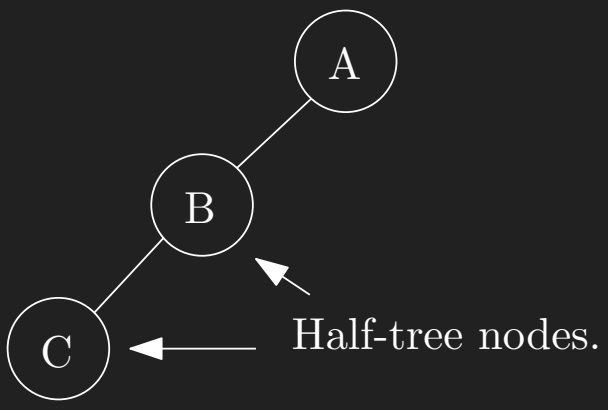
- ***Definición de "Two tree or strictly binary tree":*** *is a binary tree where each node is either a leaf, or they are having both children. Meaning no half-leaf nodes.*

Strictly binary tree.



No half-leaf nodes.

Not a strictly binary tree.

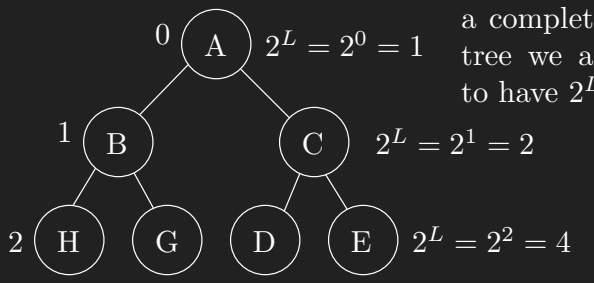


12.5 Complete binary tree / full tree

- ***Definición de "Complete binary tree / full tree ":*** *A complete binary tree is a 2-tree where all leaves must reside at the same level. OR, in a complete binary tree at any leaf k, there are always $2k$ nodes.*

Example of complete binary tree

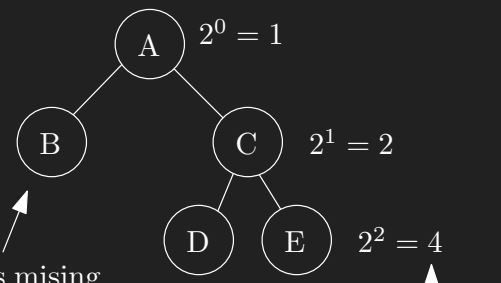
Each level is full.



At any level in a complete binary tree we are going to have 2^L nodes.

Example of incomplete binary tree

Every level is not full.



Two leafs mising in the B node sub-tree.

There are not 4 nodes in level 2.

- The total nodes in a complete binary tree is calculated by:

$$t_n = 2^{h+1} - 1$$

or

$$t_n = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^h$$

Where the h is the height.

  - In the example above is $2^{2+1} - 1 = 8 - 1 = 7$.

- To calculate the number of non-leaves.

$$\text{non-leaves } = 2^h - 1$$

- Total leaves:

$$\text{total-leaves } = 2^h$$

12.6 How to traverse a binary tree

There are three strategies for traversing a binary tree:

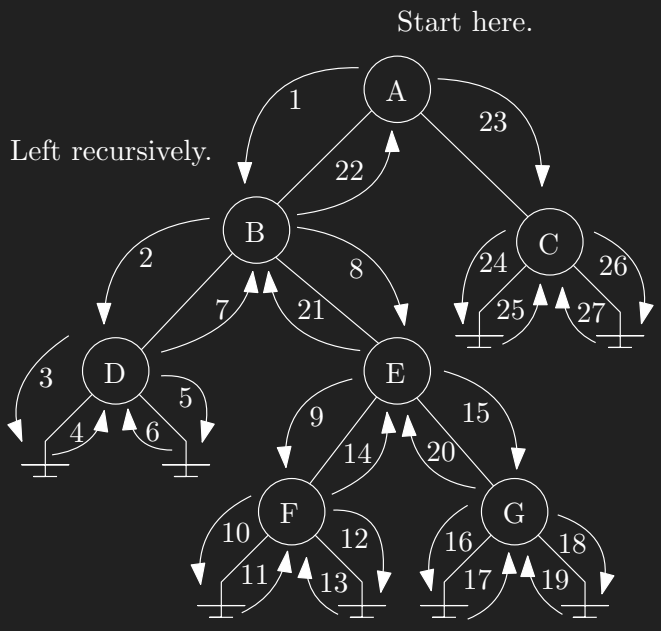
1. In-order traversal.
2. Pre-order traversal.
3. Post-order traversal.

In a binary trees it's impossible to traverse linearly, they are not like arrays or linked lists in the sense where you can do a loop and traverse. Trees are not linear data structures and in order to visit every node of the tree at least once we need to have proper traversal strategies. Whenever we traverse a binary tree we must do it recursively.

12.6.1 In-order traversal strategy

This strategy consists of implementing a recursive algorithm. This algorithm considers every current node as a sub-tree in itself.

1. Traverse the left sub-tree using in-order routine.
2. Access root.
3. Traverse right sub-tree using in order routine.



Order of nodes visited: D at step 4, B at step 7, F at step 11, E at step 14, G at step 17, A at step 22, C at step 25.

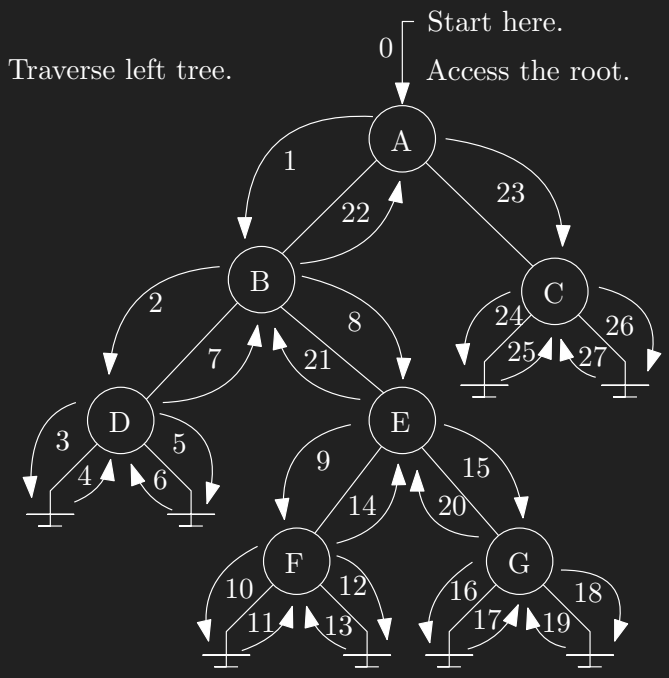
$$D, B, F, E, G, A, C$$

Insight: left $\rightarrow$ root $\rightarrow$ right.

12.6.2 Pre-order traversal strategy

It's different from the in-order because here the root stays constant, this algorithm doesn't consider each node the root.

1. Access the root.
2. Traverse left sub-tree using the pre-order algorithm using recursion.
3. Traverse right-sub-tree using pre-order.



Order of nodes visited: A accessed at step 0, B accessed at step 1, D accessed at step 2, E accessed at step 8, F accessed at step 9, G accessed at step 15, C accessed at step 23.

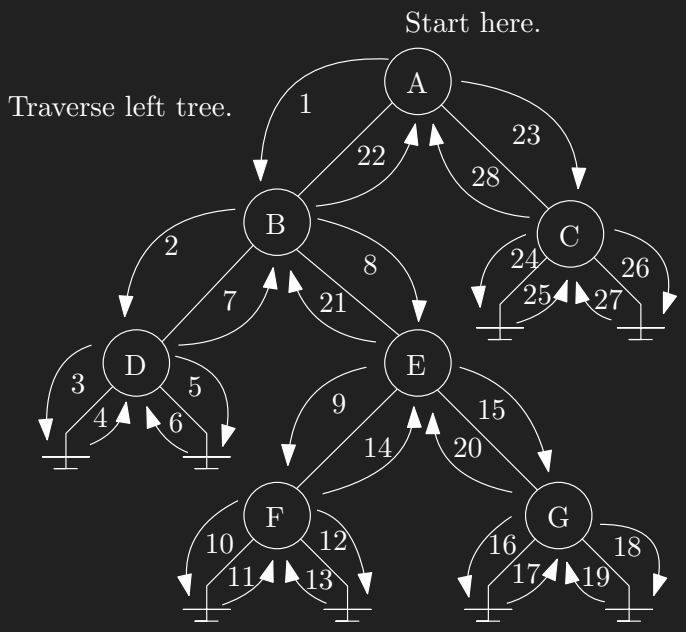
$$A, B, D, E, F, G, C$$

Insight: root $\rightarrow$ left $\rightarrow$ right.

12.6.3 Post-order traversal strategy

In this implementation the root is accessed at the end. The position where we access the root is at the end. This is also recursive.

1. Traverse the left-sub-tree using post-order.
2. Traverse the right-sub-tree using post-order.
3. Access the root.



Order of nodes visited: D accessed at step 6, F accessed at step 13, G accessed at 19, E accessed at step 20, B accessed at step 21, C accessed at 27, A accessed at step 28.

$$D, F, G, E, B, C, A$$

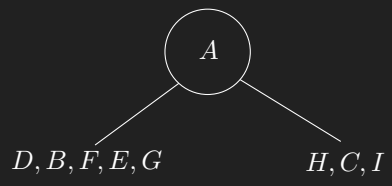
Insight: left $\rightarrow$ right $\rightarrow$ root.

12.7 Constructing a binary tree from a given traversal list

- In-order traversal list: $D, B, F, E, G, A, H, C, I$
- Pre-order traversal list: $A, B, D, E, F, G, C, H, I$
- Post-order traversal list: $D, F, G, E, B, H, I, C, A$

12.7.1 Developing the tree using the in-order and pre-order lists

- The pre-order traversal, we access the root of the entire tree first. Thus, A will be the root of the tree.
- In the in-order traversal, we locate A, we know that everything to the left of A in the in-order traversal list is the left sub-tree and everything to the right is the right sub-tree.
  - With this information we know that:



- Know exmining the elements of the left subtree, we have D, B, F, E, G Notice that these elements exist

consecutively after the A in the pre-order traversal, not ordered but consecutively:

$$\text{Pre-order traversal list: } A, \overbrace{B, D, E, F, G}^{\text{L. sub-tree}}, C, H, I$$

- By examining the rest, this means the remaining letters: C, H, I are the right sub-tree.

$$\text{Pre-order traversal list: } A, B, D, E, F, G, \overbrace{C, H, I}^{\text{R. sub-tree}}$$

- Notice in the pre-order traversal, the first letter of the left sub-tree is B, this means the B is the root of the left sub-tree.
- Now, locate B in the in-order traversal list, anything to the left of B is the left sub-tree of root B and anything to the right is the right sub-tree of root B.

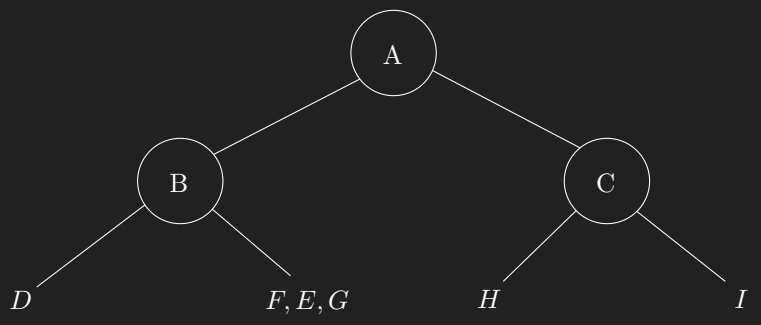
$$\text{Pre-order traversal list of the L. sub-tree of root B: } B, D, E, F, G$$

$$\text{In-order traversal list of the L. sub-tree of root B: } \overbrace{D}^{\text{R.sub-tree}}, B, \overbrace{F, E, G}^{\text{L. sub-tree}}$$

- Notice in the pre-order traversal, the first letter of the right sub-tree is C, thus this is the root of the right sub-tree.
- Now, locate C in the in-order traversal list, anything to the left is the left sub-tree, anything to the right is the right sub-tree.
- With the information thus discovered we know the following:

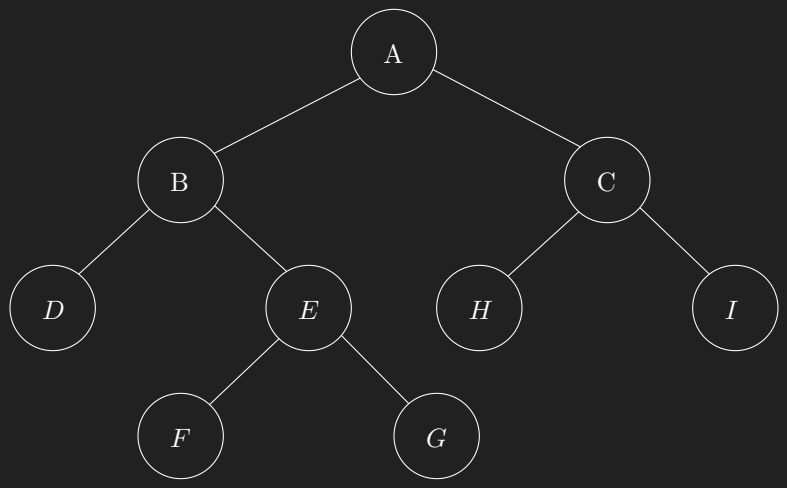
$$\text{Pre-order traversal list of the R. sub-tree of root C: } \overbrace{C}^{\text{Root}}, H, I$$

$$\text{In-order traversal list of the R. sub-tree of root C: } \overbrace{H}^{\text{L. sub-tree}}, C, \overbrace{I}^{R.sub-tree}$$



- Notice the pre-order traversal $A, B, D, E, F, G, C, H, I$ after the D the next is the E this means this is the root of the right sub-tree of the tree of root B.
- Notice with the in-order traversal list corresponding to the right sub-tree of tree of root B: F, E, G, E is the root, this means anything to the left is the left sub-tree, anything to the right is the right sub-tree of tree root E.
- Repeat the process for the C right sub-tree.

- The end result is:



12.7.2 Developing a tree from in-order and post-order traversal lists

- The root of the tree is the very last element, in this case it's A .
- Track A in in-order traversal. Left: D, B, F, E, G, Right: H, C, I.
- Identify the left and right tree in the in-order traversal list, then track the contents of both trees to the post order traversal list. Right: H, I, C Left: D, F, G, E, B read from left to right.
- The right sub-tree traversal list of root A's last letter is the root of the Right sub-tree. Right: H, I, C so C is the root of the right sub-tree. Checking in the in-order traversal list we find the root C, anything to the left is the left sub-tree, and the same for the right. In-order traversal: H, C, I, H is the left sub-tree and I is the right-sub-tree.
- The left sub-tree traversal list of root A's last letter is the root of the left sub-tree. Left: D, F, G, E, B so root is B. Checking them in the in-order traversal list we find the root B in D, B, F, E, G to the left we have the left sub-tree, same for the right.
- Repeating the process should land us in the same result.

12.8 How to define a structure of a Node for a binary tree

We need three elements:

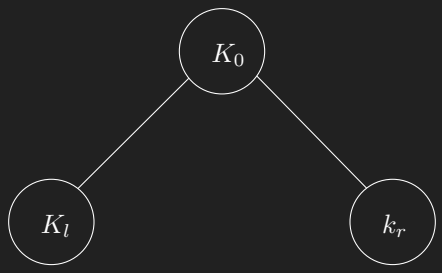
1. A search key:
2. Left-pointer: pointing to the left sub-tree, if no sub-tree point to null.
3. Right-pointer: pointing to the right sub-tree, if no sub-tree point to null.

12.9 Binary search tree

- To apply binary search on list/array elements, the list or array must be sorted.
- Sorting algorithms worse case complexity of quicksort or merge sort is $O(n * \log(n))$.
- The worse case complexity of binary search is $O(\log(n))$.

- The overall instead of sorting with quick sort or merge sort and getting complexity $O(n * \log(n))$ performance, using a binary search tree the elements will be accomodated while they are being inserted thus the complexity will always be $O(\log(n))$.

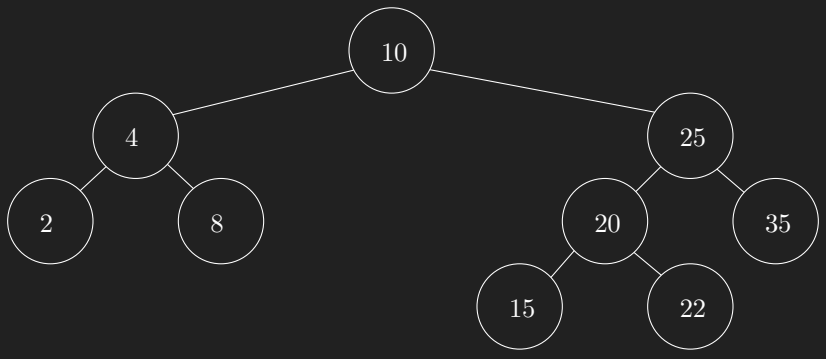
Binary search tree definition:



$$K_l < K_0 < K_r$$

Example:

$$\texttt{Keys:}\ 10, 25, 4, 20, 2, 8, 35, 15, 22$$



12.10 Binary tree implementation

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

typedef struct Node {
    int key;
    struct Node *right;
    struct Node *left;
} Node;

typedef struct BinarySearchTree {
    Node *root;
} BinarySearchTree;

void init(BinarySearchTree *bst){
    bst->root = NULL;
}

```

```c
void insert(Node **root_ptr, int key){
    /* Keep in mind every node is a root of it's own sub-tree.
       This root might not be the root of the entire tree */
    if (*root_ptr == NULL) {
        *root_ptr = (Node*)malloc(sizeof(Node));
        if (*root_ptr == NULL){
            printf("Error allocating space.\n");
            exit(1);
        }
        (*root_ptr)->key = key;
        (*root_ptr)->left = (*root_ptr)->right = NULL;
    } else if ( key < (*root_ptr)->key ) { // left
        insert(&((*root_ptr)->left),key);
    } else if ( key > (*root_ptr)->key  ) { // right
        insert(&((*root_ptr)->right),key);
    } else {
        printf("Failed, key already exists in the tree.");
    }
}

void inorder(Node *root_ptr){
    if (root_ptr != NULL){
        inorder(root_ptr->left); // inorder traversal to the leftest side
        printf("%d ",root_ptr->key); // access
        inorder(root_ptr->right); // inorder traversal to the rightest side
    }
    else {
        return;
    }
}

void preorder(Node *root_ptr){
    if (root_ptr != NULL){
        printf("%d ",root_ptr->key); // access
        preorder(root_ptr->left); // inorder traversal to the leftest side
        preorder(root_ptr->right); // inorder traversal to the rightest side
    }
    else {
        return;
    }
}

void postorder(Node *root_ptr){
    if (root_ptr != NULL){
        postorder(root_ptr->left); // inorder traversal to the leftest side
        postorder(root_ptr->right); // inorder traversal to the rightest side
        printf("%d ",root_ptr->key); // access
    }
    else {
        return;
    }
}

Node *binary_search(Node *root_ptr, int target) {
```

```
    if (root_ptr == NULL){
        return NULL;
    } else if (root_ptr->key == target){
        return root_ptr;

    } else if (target < root_ptr->key) {
        return binary_search(root_ptr->left,target);
    } else if (target > root_ptr->key){
        return binary_search(root_ptr->right,target);
    }
}

void menu() {
    printf("1. Insert operation.\n");
    printf("2. In-order traversal.\n");
    printf("3. Pre-order traversal.\n");
    printf("4. Post-order traversal.\n");
    printf("5. Binary search.\n");
    printf("6. Exit.\n");
}


int main() {
    int choice, key, target, quit=0;
    Node *target_node = NULL;
    BinarySearchTree bst;
    init(&bst);
    insert(&bst.root,0);
    insert(&bst.root,10);
    insert(&bst.root,45);
    insert(&bst.root,98);
    insert(&bst.root,120);
    menu();
    while (!quit) {
        printf("Enter the choice: ");
        scanf("%d",&choice);
        switch (choice) {
            case 1: printf("Enter the key:");
                    scanf("%d",&key);
                    insert(&bst.root,key);
                    break;
            case 2: inorder(bst.root);
                    printf("\n");
                    break;
            case 3: preorder(bst.root);
                    printf("\n");
                    break;
            case 4: postorder(bst.root);
                    printf("\n");
                    break;
            case 5: printf("Enter the target: ");
                    scanf("%d",&target);
                    target_node = binary_search(bst.root, target);
                    target_node == NULL?
```

```
                                printf("Not found.\n") :
                                printf("%d found at %p\n",target_node->key,target_node);
                        break;
                case 6: exit(0);
                        break;
                default: printf("Invalid choice.\n");
                        break;
        }
    }
    return 0;
}


/* Output:
1. Insert operation.
2. In-order traversal.
3. Pre-order traversal.
4. Post-order traversal.
5. Binary search.
6. Exit.
Enter the choice: 2
0 10 45 98 120
Enter the choice: 5
Enter the target: 23
Not found.
Enter the choice: 120
Invalid choice.
Enter the choice: 5
Enter the target: 120
120 found at 0x800062c10
Enter the choice: 6
*/
```

Chapter 13

Heap

13.1 Introduction

- Another comparison based sorting algorithm.
- Worst case complexity of $O(n \log(n))$.
- The space complexity of heap sort is contant $O(1)$.
- Heap sort is a improved version of selection sort that uses the heap data structure.
- Consists of: Build a heap with the unsorted data, then delete the elements from unsorted, and order them in another heap.

13.2 Almost complete binary

A binary tree in which nodes can only me added from left to right.

13.3 Representing an almost complete binary tree as an array

- For any parent i the children are:
  - Left: $2i$
  - Right: $2i + 1$
- For any child node k, the parent is $\lfloor k/2 \rfloor$.

13.4 Formal definition of heap

Maxheap:

- The largest value is stored in the root.

Minheap:

- The smallest value is stored in the root.