Lecture 5

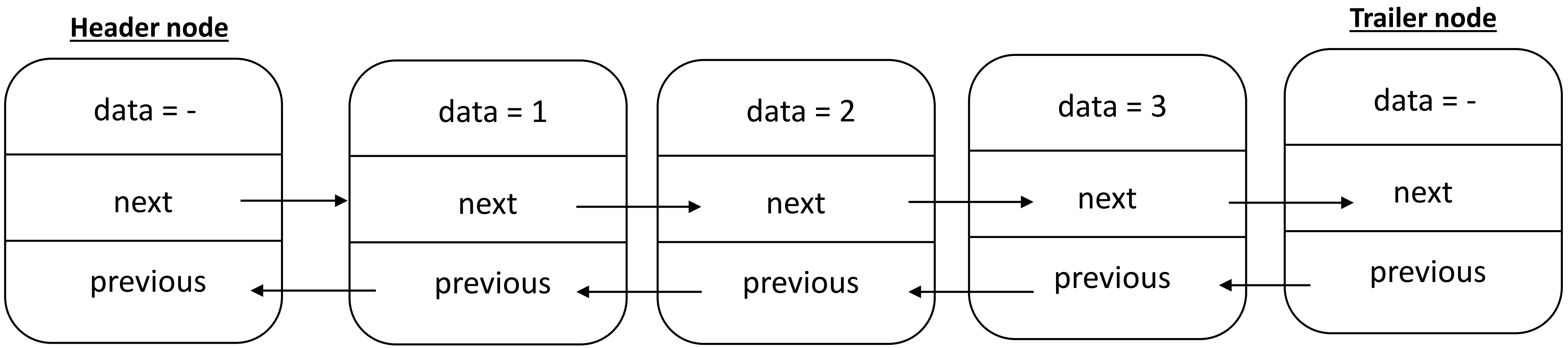
Sentinels, Circular Linked Lists, Stacks

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Sentinel Nodes

- Special "dummy" nodes added at both ends of a doubly linked list. Does not store elements.
- Header Node
  - Placed just before the first node (head).
  - The next pointer points to the first actual node.
- Trailer Node
  - Placed just after the last node (tail).
  - The previous pointer points to the last actual node.



Sentinel Nodes in Doubly Linked Lists

- Special "dummy" nodes added at both ends of a doubly linked list. Does not store elements.

```c
typedef struct intNode {
    int data;
    struct intNode *next;
    struct intNode *previous;
} intNode;

typedef struct intDoublyList {
    intNode *header;
    intNode *tailer;
    int elemcount;
} intDoublyList;
```

```c
void initDoublyList(intDoublyList *list) {
    list->header = (intNode *)malloc(sizeof(intNode));
    list->tailer = (intNode *)malloc(sizeof(intNode));
    list->header->next = list->tailer;
    list->tailer->previous = list->header;
    list->header->previous = NULL;
    list->tailer->next = NULL;
    list->elemcount = 0;
}
```

```c
void addBack(intDoublyList *list, int new_element) {
    intNode *newnode = (intNode *)malloc(sizeof(intNode));
    newnode->data = new_element;

    newnode->next = list->tailer;
    newnode->previous = list->tailer->previous;

    list->tailer->previous->next = newnode;
    list->tailer->previous = newnode;

    list->elemcount++;
}
```

```c
void removeBack(intDoublyList *list) {
    if (list->elemcount > 0) {
        intNode *old = list->tailer->previous;

        old->previous->next = list->tailer;
        list->tailer->previous = old->previous;

        free(old);
        list->elemcount--;
    }
}
```

Sentinel Nodes in Doubly Linked Lists

```c
void addAtPosition(intDoublyList *list, int new_element, int position) {
    if (position < 0 || position > list->elemcount) {
        return;
    }

    intNode *newnode = (intNode *)malloc(sizeof(intNode));
    newnode->data = new_element;

    intNode *position_pointer = list->header->next;

    for (int index = 0; index < position; index++) {
        position_pointer = position_pointer->next;
    }

    newnode->next = position_pointer;
    newnode->previous = position_pointer->previous;
    position_pointer->previous->next = newnode;
    position_pointer->previous = newnode;

    list->elemcount++;
}
```

```c
void removeAtPosition(intDoublyList *list, int position) {
    if (position < 0 || position >= list->elemcount) {
        return;
    }

    intNode *position_pointer = list->header->next;

    for (int index = 0; index < position; index++) {
        position_pointer = position_pointer->next;
    }

    position_pointer->previous->next = position_pointer->next;
    position_pointer->next->previous = position_pointer->previous;

    free(position_pointer);
    list->elemcount--;
}
```

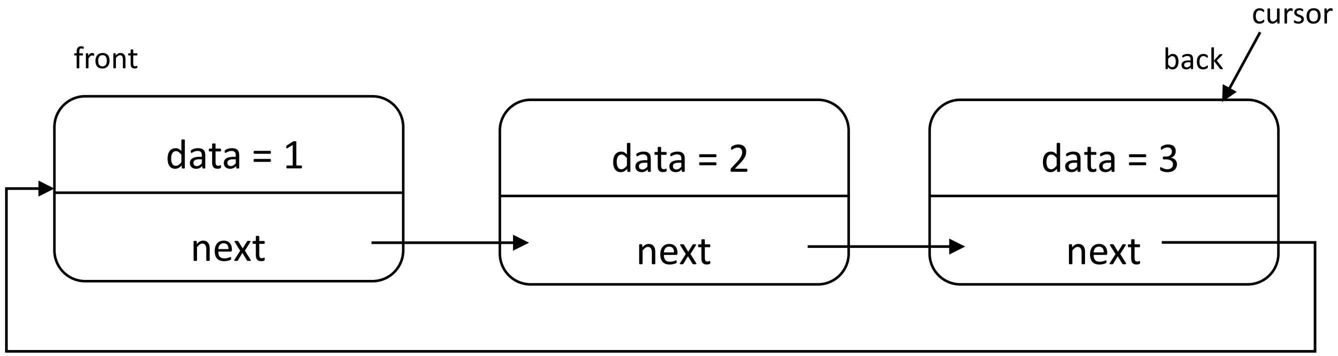
Circular Linked List

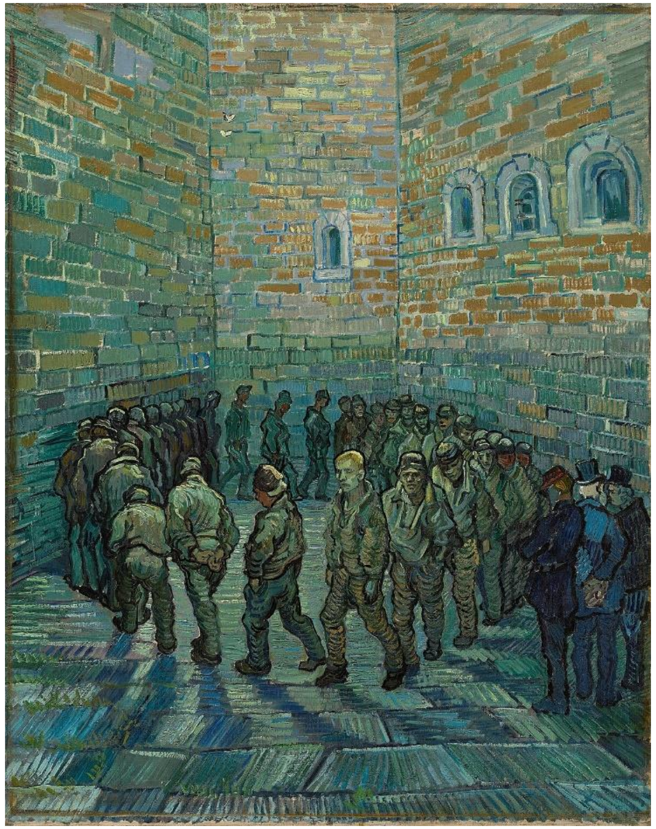
- In a circularly linked list, instead of having a head or tail, the nodes are connected in a loop.
- When navigating through a circularly linked list by the next pointers from any given node, we will pass through every node and return to the starting node, completing a full cycle.

- It's still necessary to designate a specific node as a reference point, known as the cursor.
  - **Back:** The element referenced by the cursor.
  - **Front:** The next element

Anytime the Circular Linked List could be transformed into a linked list by:

- Assigning NULL to the end element's next pointer.
- Creating a head pointer to show the front element.

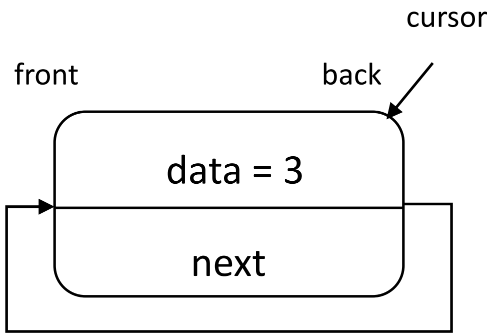




Prisoners' Round, Vincent Van Gogh

Circular Linked List: Adding & Removing

- If the list has no elements, its firstly added node should point itself.




```c
CircularList mylist;
initCircularList(&mylist);
addEnd(&mylist, 1);

Node *ptr = mylist.cursor;

for (int i = 0; i < 100; i++) {
    printf("%d\n", ptr->data);
    ptr = ptr->next;
}
```

```
1
1
1
1
1
1
1
1
1
...
```

```c
typedef struct Node {
    int data;
    struct Node *next;
} Node;

typedef struct CircularList {
    Node *cursor;
    int elemcount;
} CircularList;
```

```c
void initCircularList(CircularList *list) {
    list->cursor = NULL;
    list->elemcount = 0;
}
```

```c
void removeFront(CircularList *list) {
    if (list->cursor == NULL) return;

    Node *to_delete = list->cursor->next;

    if (to_delete == list->cursor) {
        list->cursor = NULL;
    } else {
        list->cursor->next = to_delete->next;
    }

    free(to_delete);
    list->elemcount--;
}
```

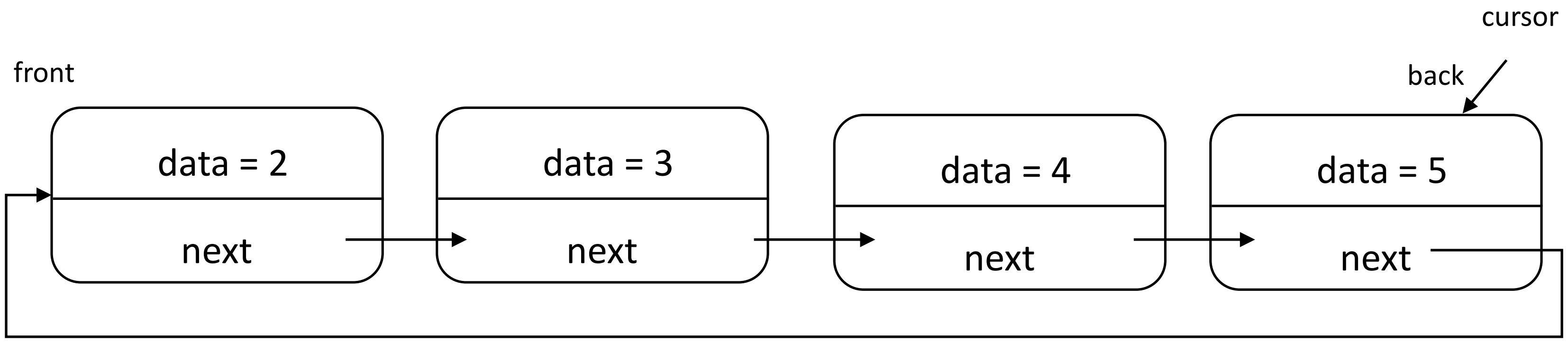
```c
void addEnd(CircularList *list, int new_element) {
    Node *newnode = (Node *)malloc(sizeof(Node));
    newnode->data = new_element;

    if (list->cursor == NULL) {
        newnode->next = newnode;
        list->cursor = newnode;
    } else {
        newnode->next = list->cursor->next;
        list->cursor->next = newnode;
        list->cursor = newnode;
    }

    list->elemcount++;
}
```

Circular Linked List

- We could endlessly iterate the list.

```c
CircularList mylist;
mylist.cursor = NULL;

for (int i = 5; i > 0; i--)
    addEnd(&mylist, i);
removeFront(&mylist);
Node *ptr = mylist.cursor;

for (int i = 0; i < 100; i++) {
    printf("%d\n", ptr->next->data);
    ptr = ptr->next;
}
```

```
2
3
4
5
2
3
4
5
...
```




```c
int back(CircularList *list) {
    if (list->cursor != NULL) {
        return list->cursor->data;
    }
    return -1;
}
```

```c
int front(CircularList *list) {
    if (list->cursor != NULL) {
        return list->cursor->next->data;
    }
    return -1;
}
```

```c
void advance(CircularList *list) {
    if (list->cursor != NULL) {
        list->cursor = list->cursor->next;
    }
}
```

How to reverse?

- According to the node connections, there are two main strategies to reverse linked lists.

Singly & Circular Linked List

- Repeatedly take and the first element of the list and store them in a temporary list.
- Copy each element from the temporary list to the back of the original list.

```c
void addFront(CircularList* list, int new_element) {
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->data = new_element;

    if (list->cursor == NULL) {
        newnode->next = newnode;
        list->cursor = newnode;
    } else {
        newnode->next = list->cursor->next;
        list->cursor->next = newnode;
        advance(list);
    }
    list->elemcount++;
}
```

```c
void reverse(CircularList* list) {
    if (list->cursor == NULL) return;

    CircularList temp;
    temp.cursor = NULL;
    temp.elemcount = 0;

    int length = list->elemcount;
    for (int i = 0; i < length; i++) {
        int frontValue = front(list);
        addFront(&temp, frontValue);
        removeFront(list);
    }

    for (int i = 0; i < length; i++) {
        int frontValue = front(&temp);
        addEnd(list, frontValue);
        removeFront(&temp);
    }
}
```

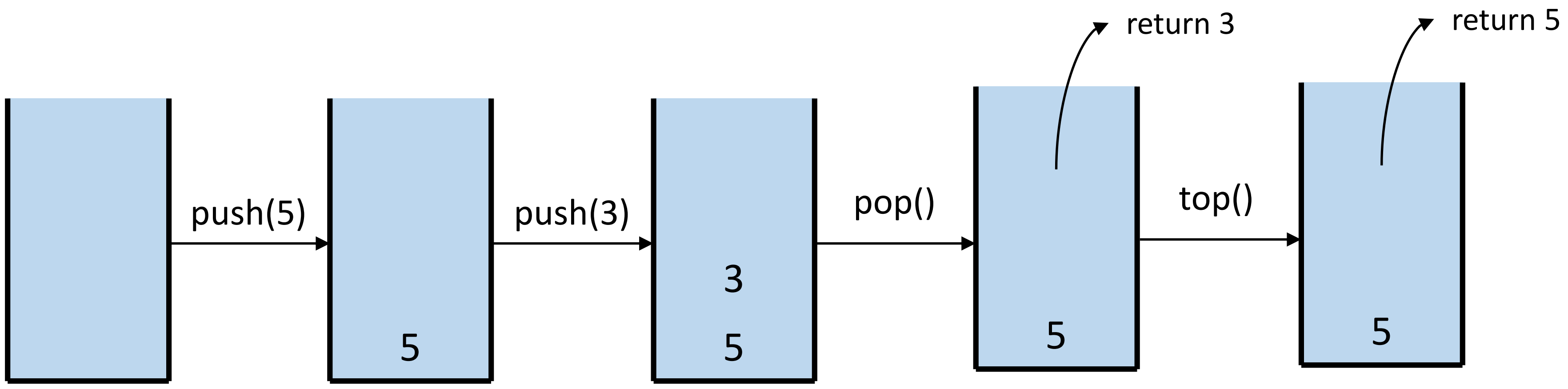
Doubly Linked List

- Swap the next and previous pointers

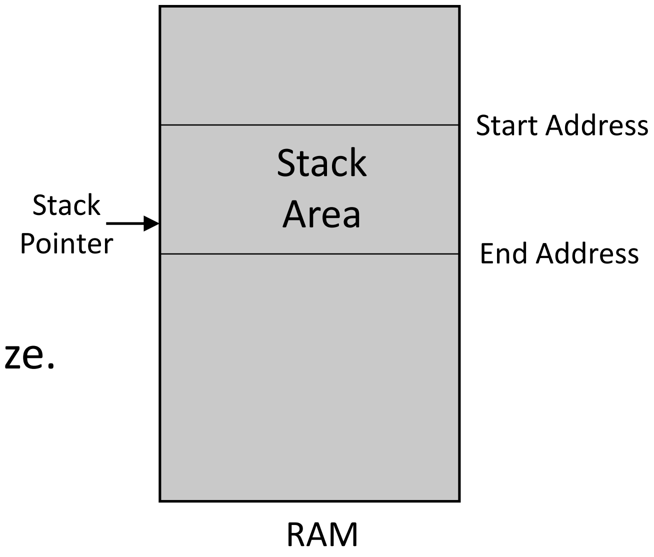
```c
void reverseDoublyList(DoublyList *list) {
    DNode *ptr = list->head;
    DNode *temp = NULL;

    while (ptr != NULL) {
        temp = ptr->previous;
        ptr->previous = ptr->next;
        ptr->next = temp;
        ptr = ptr->previous;
    }

    if (temp != NULL) {
        list->tail = list->head;
        list->head = temp->previous;
    }
}
```

Stack

- A stack operates as a storage for objects, where they are added and taken out following the last-in, first-out (LIFO) strategy.
- There are three main operations:
    - **Push:** Insert an element at the top of the stack.
    - **Pop:** Remove the top element of the stack and return the value.
    - **Top:** Return the reference or the value of the top element.



Stack implementation with an array



- A simple implementation of a stack can be achieved using an array with a predefined size.
- Similar to the stack usage of a microcomputer inside the RAM.

```c
typedef struct {
    int data[LIMIT];
    int elemcount;
} Stack;

void initStack(Stack* stack) {
    stack->elemcount = 0;
}
```

```c
int pop(Stack* stack) {
    if (stack->elemcount > 0) {
        stack->elemcount--;
        return stack->data[stack->elemcount];
    } else {
        printf("Stack underflow\n");
        return -1;
    }
}
```

```c
int isEmpty(Stack* stack) {
    return stack->elemcount == 0;
}
```

```c
void push(Stack* stack, int new_element) {
    if (stack->elemcount < LIMIT) {
        stack->data[stack->elemcount] = new_element;
        stack->elemcount++;
    } else {
        printf("Stack overflow\n");
    }
}
```

```c
int top(Stack* stack) {
    if (stack->elemcount > 0) {
        return stack->data[stack->elemcount - 1];
    } else {
        printf("Stack is empty\n");
        return -1;
    }
}
```

```c
Stack mystack;
initStack(&mystack);

for (int i = 5; i > 0; i--)
    push(&mystack, i);

printf("%d\n", pop(&mystack));
printf("%d\n", pop(&mystack));
printf("%d\n", pop(&mystack));
printf("%d\n", top(&mystack));
printf("%d\n", top(&mystack));
```

```
1
2
3
4
4
```

Stack implementation with a linked list

- Using linked lists to store the data, we may change the size dynamically.

```c
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct {
    Node* head;
    int elemcount;
} Stack;
```

```c
void initStack(Stack* stack) {
    stack->head = NULL;
    stack->elemcount = 0;
}
```

```c
void push(Stack* stack, int new_element) {
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->data = new_element;
    newnode->next = stack->head;
    stack->head = newnode;
    stack->elemcount++;
}
```

```c
int pop(Stack* stack) {
    if (stack->head == NULL) {
        printf("Stack underflow\n");
        return -1;
    }

    Node* temp = stack->head;
    int popped_data = temp->data;
    stack->head = stack->head->next;
    free(temp);
    stack->elemcount--;

    return popped_data;
}
```

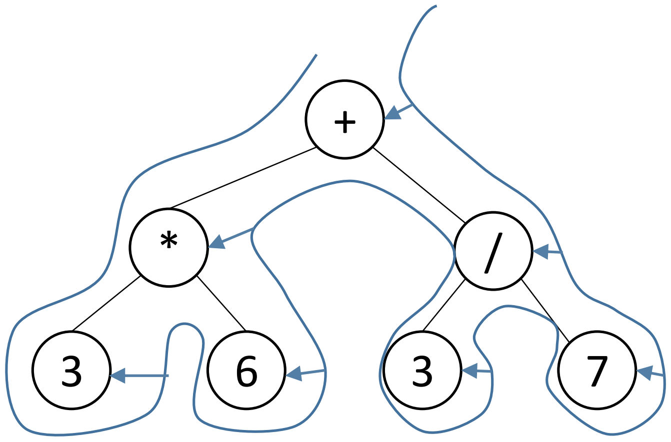
```c
int top(Stack* stack) {
    if (stack->head == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    return stack->head->data;
}
```

```c
Stack mystack;
initStack(&mystack);

for (int i = 5; i > 0; i--)
    push(&mystack, i);

printf("%d\n", pop(&mystack));
printf("%d\n", pop(&mystack));
printf("%d\n", pop(&mystack));
printf("%d\n", top(&mystack));
printf("%d\n", top(&mystack));
```

```
1
2
3
4
4
```

Ex: Mathematical Calculations of a Postfix Expression

- There are three different notations used to write expressions in mathematics.
  - **Infix:** Operators are placed between the operands.
  - **Prefix:** Operators precede the operands.
  - **Postfix:** Operators follow the operands.

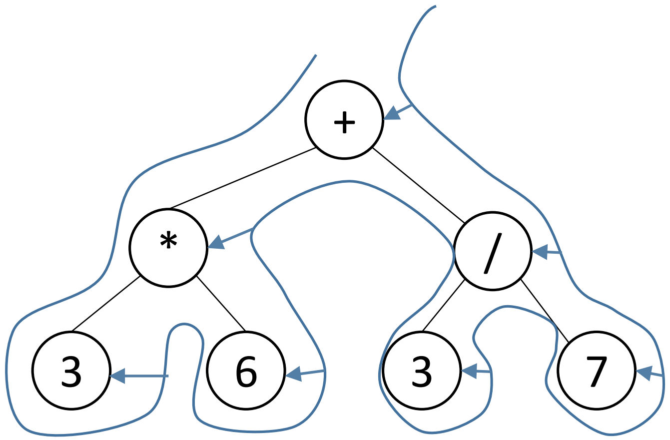


Infix: (3*6) + 3/7
Postfix: 3 6 * 3 7 / +
Prefix: + * 3 6 / 3 7

- We can employ a stack to calculate the expressions efficiently. Let's consider the postfix notation:
  - Process the expression in from left to right.
  - If the character is an operand, push it onto the stack.
  - If the character is an operation:
    - Pop the operands
    - Perform the operation
    - Push the result to the stack.
  - After the process, the stack will contain only the result.

Ex: Mathematical Calculations of a Postfix Expression

```c
float evaluatePostfix(char* math_exp[], int size) {
    Stack mystack;
    initStack(&mystack);

    for (int i = 0; i < size; i++) {
        if (math_exp[i][0] != '+' && math_exp[i][0] != '-' && math_exp[i][0] != '*' && math_exp[i][0] != '/') {
            float value = atof(math_exp[i]);
            push(&mystack, value);
        } else {
            float op1 = pop(&mystack);
            float op2 = pop(&mystack);
            float result;

            if (strcmp(math_exp[i], "/") == 0)
                result = op2 / op1;
            else if (strcmp(math_exp[i], "+") == 0)
                result = op2 + op1;
            else if (strcmp(math_exp[i], "*") == 0)
                result = op2 * op1;
            else if (strcmp(math_exp[i], "-") == 0)
                result = op2 - op1;

            push(&mystack, result);
        }
    }

    return pop(&mystack);
}
```



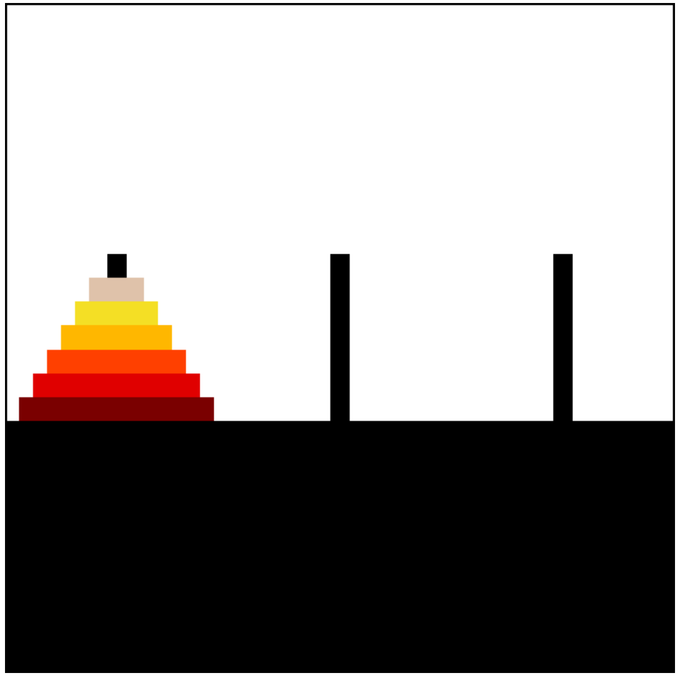
=18.4286

Ex: Towers of Hanoi

- The goal is to move all disks from initial position to another position.
- A larger disk may not be placed on top of a smaller disk.

```c
int diskcount = 3;
Stack stacks[3];

for (int i = 0; i < 3; i++)
    initStack(&stacks[i]);

for (int i = diskcount; i > 0; i--)
    push(&stacks[0], i);

while (1) {
    int src, dst;

    printf("Source (0-2): ");
    scanf("%d", &src);
    printf("Destination (0-2): ");
    scanf("%d", &dst);

    if (stacks[dst].elemcount == 0 || top(&stacks[src]) < top(&stacks[dst])) {
        int selected_disk = pop(&stacks[src]);
        push(&stacks[dst], selected_disk);
    }
    if (stacks[2].elemcount == diskcount) {
        printf("Puzzle solved!\n");
        break;
    }
}
```

```
Source (0-2): 0
Destination (0-2): 2
Source (0-2): 0
Destination (0-2): 1
Source (0-2): 2
Destination (0-2): 1
Source (0-2): 0
Destination (0-2): 2
Source (0-2): 1
Destination (0-2): 0
Source (0-2): 1
Destination (0-2): 2
Source (0-2): 0
Destination (0-2): 2
Puzzle solved!
```



https://en.wikipedia.org/wiki/Tower_of_Hanoi#/media/File:Iterative_algorithm_solving_a_6_disks_Tower_of_Hanoi.gif

Ex: Maze Escape

- In our maze escape game, the player (**O**) tries to reach the exit(**F**) by checking the empty tiles from four directions.

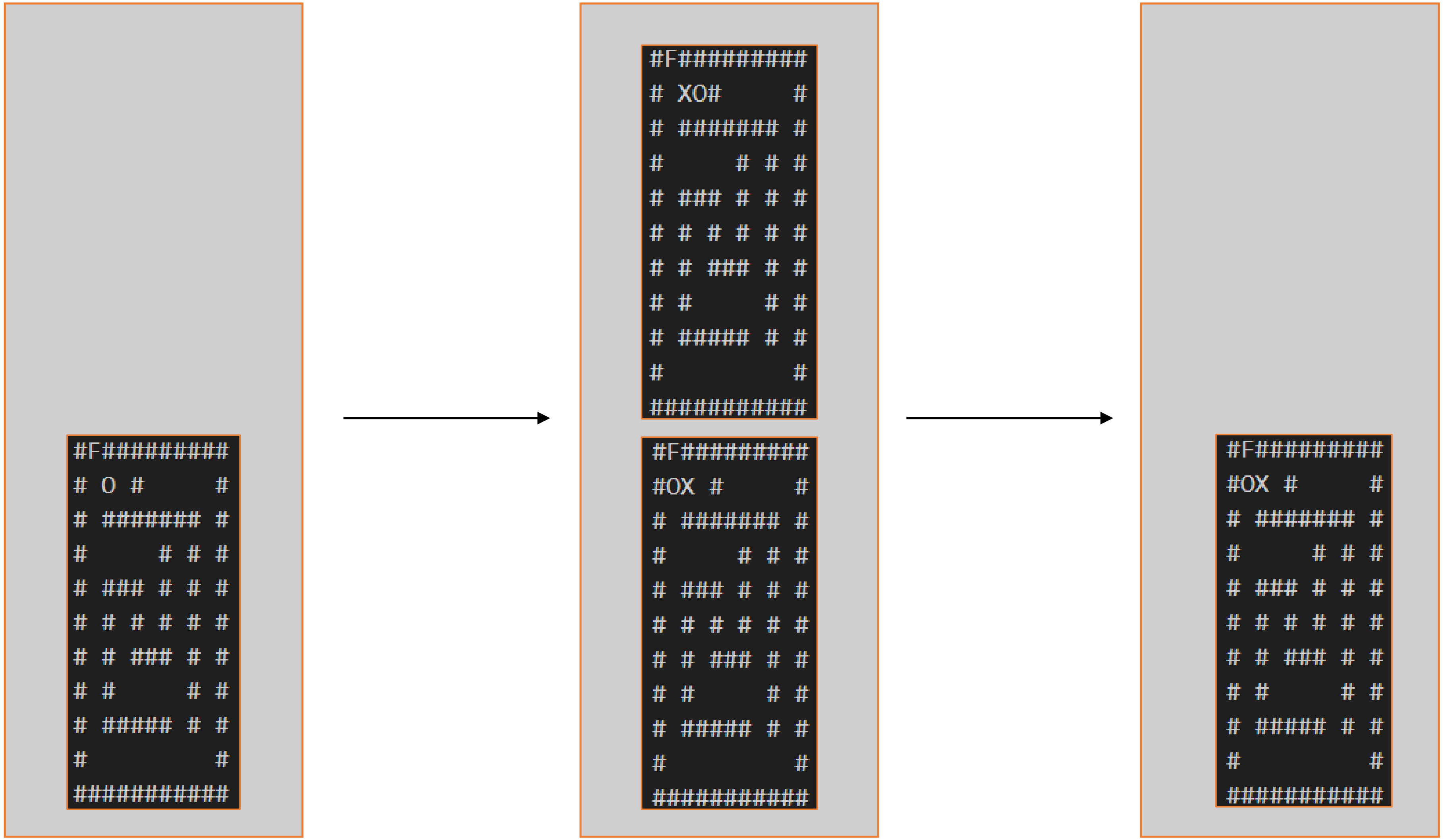
- There are so many trials and errors. To find the exact solution, stack usage is one of the most efficient options.

    - Push all the options (labyrinth states) to the stack.
    - If you have reached a dead end select the top state from the stack.

```
#F#########
#   #     #
# ####### #
#     # # #
# ### # # #
# # # # # #
# # ### # #
# #     # #
# ##### # #
#    O    #
###########
```

Ex: Maze Escape

- We can store every state of the maze in the stack.



Ex: Maze Escape

- **Task:** Escape from the maze using stack!

```c
typedef struct LabState {
    char labyrinth[11][11];
    int current_x;
    int current_y;
} LabState;

typedef struct Node {
    LabState data;
    struct Node *next;
} Node;


typedef struct Stack {
    Node *head;
    int elemcount;
} Stack;
```

```c
void push(Stack *stack, LabState e) {
    Node *newnode = (Node*)malloc(sizeof(Node));
    newnode->data = e;
    newnode->next = stack->head;
    stack->head = newnode;
    stack->elemcount++;
}

LabState pop(Stack *stack) {
    LabState result = stack->head->data;

    if (stack->head != NULL) {
        Node *old = stack->head;
        stack->head = stack->head->next;
        free(old);
        stack->elemcount--;
    }
    return result;
}
```

«labyrinth.c»

Ex: Maze Escape

- **A better solution:** Instead of full labyrinth, we may store the positions.

```c
typedef struct Position {
    int x;
    int y;
} Position;

typedef struct Node {
    Position pos;
    struct Node *next;
} Node;

typedef struct Stack {
    Node *head;
    int elemcount;
} Stack;
```

```c
void push(Stack *stack, Position pos) {
    Node *newnode = (Node*)malloc(sizeof(Node));
    newnode->pos = pos;
    newnode->next = stack->head;
    stack->head = newnode;
    stack->elemcount++;
}

Position pop(Stack *stack) {
    Position result;
    if (stack->head != NULL) {
        result = stack->head->pos;
        Node *old = stack->head;
        stack->head = stack->head->next;
        free(old);
        stack->elemcount--;
    }
    return result;
}
```

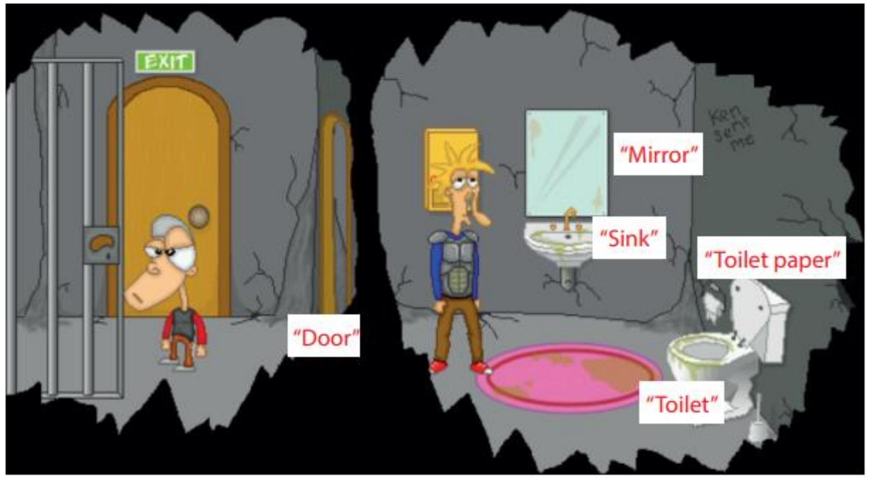
«labyrinth2.c»

Ex: Point-and-Click Adventure Game

- A genre of video game where players interact with the environment and solve puzzles.
- Players interact with NPCs, store & use items.
- In this game we have limited actions:
  - Open
  - Look At
  - Pick Up
  - Misbehave
  - Talk To

Cell

Toilet



Next Floor

https://www.adventuregamestudio.co.uk/site/games/game/554-silent-knight-chapter-1-the-mediocre-escape/