Lecture 6

Stacks and Queues

Dr. Yusuf H. Sahin
Istanbul Technical University

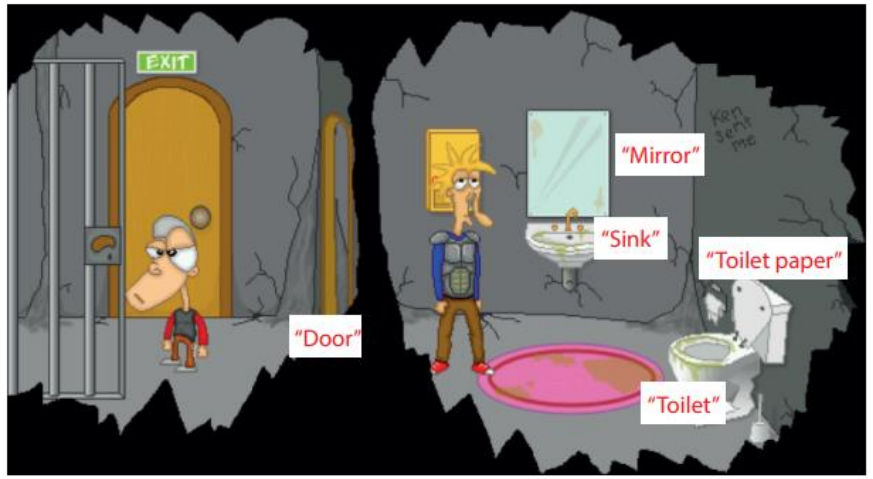
sahinyu@itu.edu.tr

Ex: Point-and-Click Adventure Game

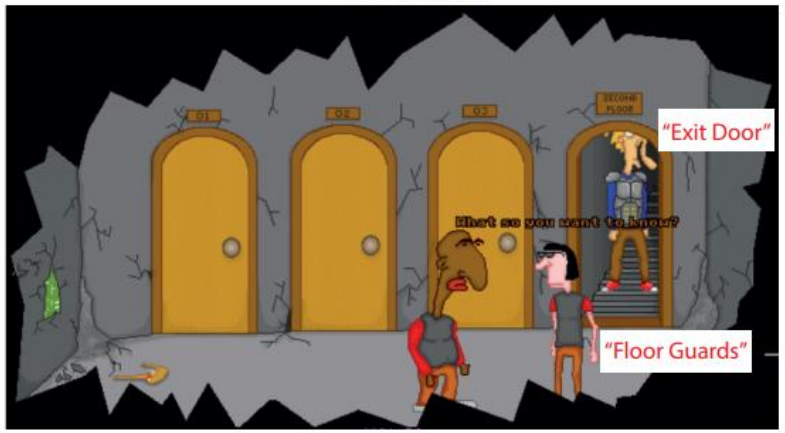
- A genre of video game where players interact with the environment and solve puzzles.
- Players interact with NPCs, store & use items.
- In this game we have limited actions:
  - Open
  - Look At
  - Pick Up
  - Misbehave
  - Talk To

Cell

Toilet



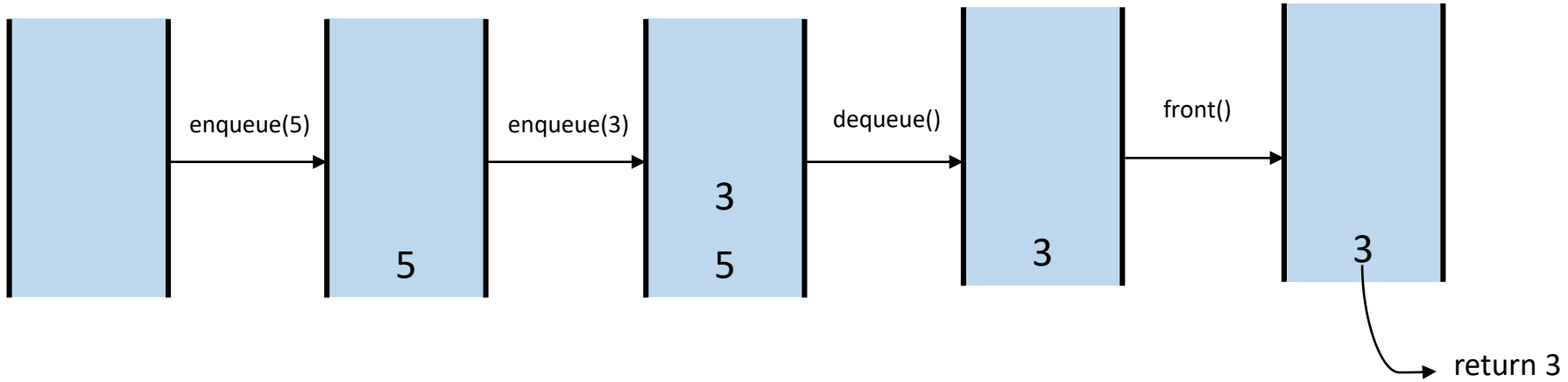
Next Floor



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

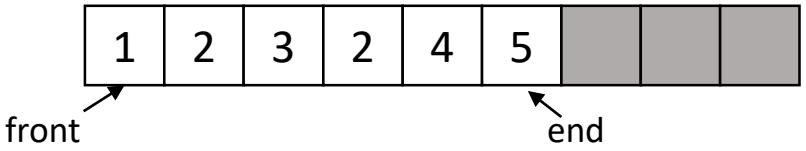
Queue

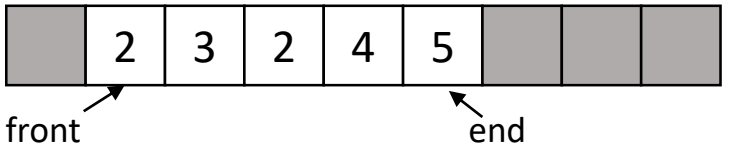
- A queue is a structure for holding elements where the addition and removal of items follow the first-in, first-out (FIFO) rule.
- Items can be added to a queue at any time, but only the item that has been in the queue for the longest duration is eligible for removal at any given moment.
- There are three main operations:
  - **Enqueue:** Insert an element at the back of the queue.
  - **Dequeue:** Remove the front element of the queue. Do not return the value!
  - **Front:** Return the reference or the value of the front element.



Implementing the Queue

- A queue could be implemented within an array…





- Limited size
- Waste of memory

- …or a linked list.

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

typedef struct Queue {
    Node* head;
    Node* tail;
    int elemcount;
} Queue;

void initQueue(Queue* q) {
    q->head = NULL;
    q->tail = NULL;
    q->elemcount = 0;
}
```

```
void enqueue(Queue* q, int new_element) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation error\n");
        return;
    }
    newNode->data = new_element;
    newNode->next = NULL;

    if (q->tail != NULL) {
        q->tail->next = newNode;
    }
    q->tail = newNode;

    if (q->head == NULL) {
        q->head = newNode;
    }

    q->elemcount++;
}
```

```
void dequeue(Queue* q) {
    if (q->head == NULL) {
        printf("Queue is empty\n");
        return;
    }

    Node* temp = q->head;
    q->head = q->head->next;
    if (q->head == NULL) {
        q->tail = NULL;
    }

    free(temp);
    q->elemcount--;
}
```

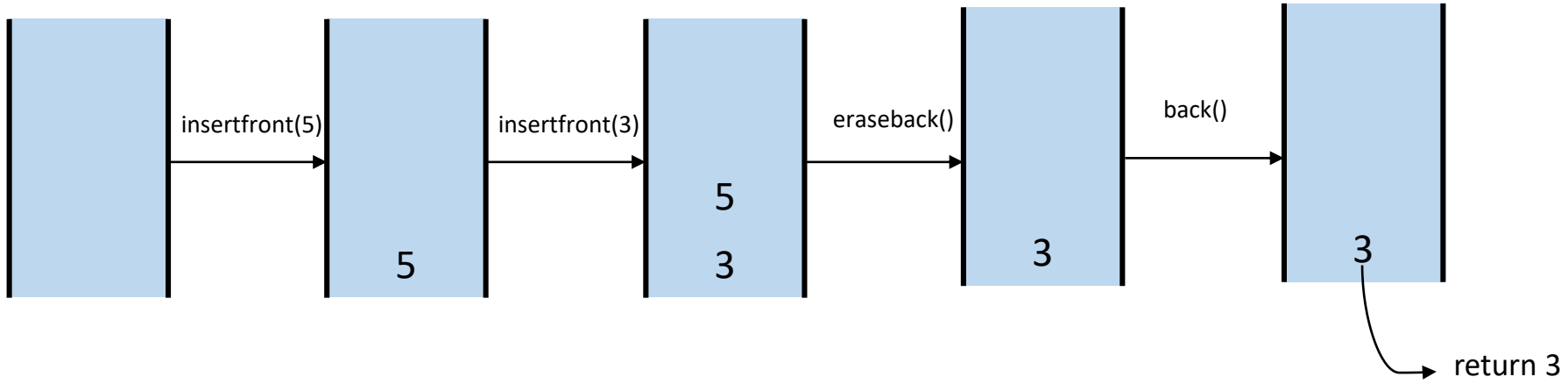
```
int front(Queue* q) {
    if (q->head == NULL) {
        printf("Queue is empty\n");
        return -1;
    }
    return q->head->data;
}
```

```
Queue myqueue;
initQueue(&myqueue);

for (int i = 5; i > 0; i--)
    enqueue(&myqueue, i);

dequeue(&myqueue);

printf("%d\n", front(&myqueue));
```

→ 4

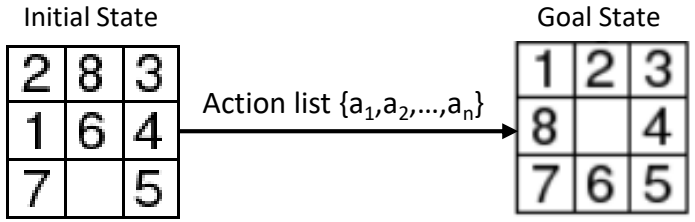
Dequeue

- A double-ended queue (dequeue) supports addition and deletion operations from the back and the front.
- There are six main operations: insertfront(T), insertback(T), erasefront(), eraseback(), front(), back()
- The most effective way to use dequeue is to use a doubly linked list.

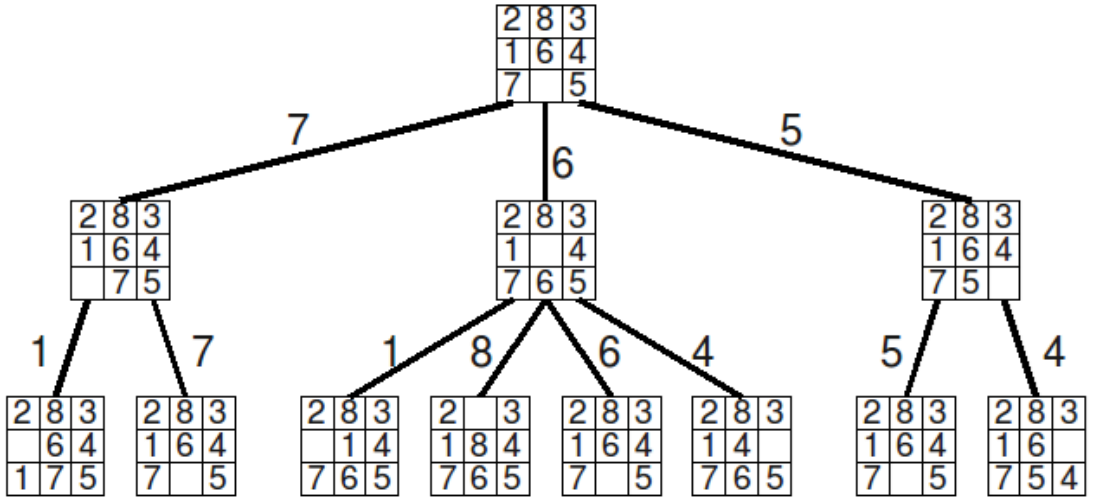


Example: State Space Search Trees

- Starting from an initial state, using a limited set of actions, find the set of actions to reach the goal state initially.



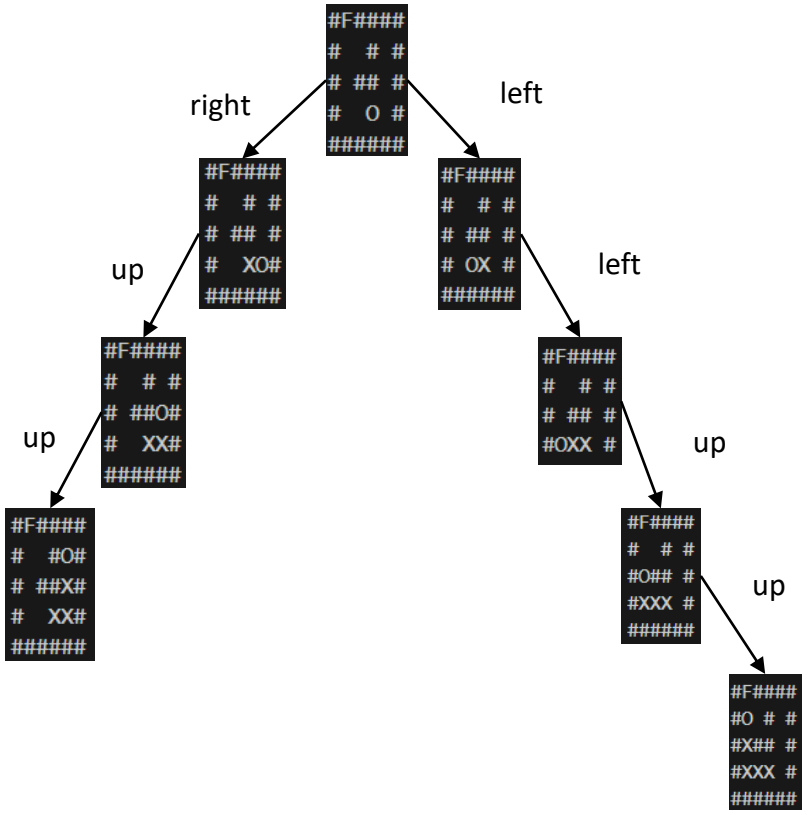
- A State Space Search Tree could be used to visualize and navigate the set of all possible states.



Borrowed from: https://www.d.umn.edu/~gshute/cs2511/slides/state_space_search/slide010.html

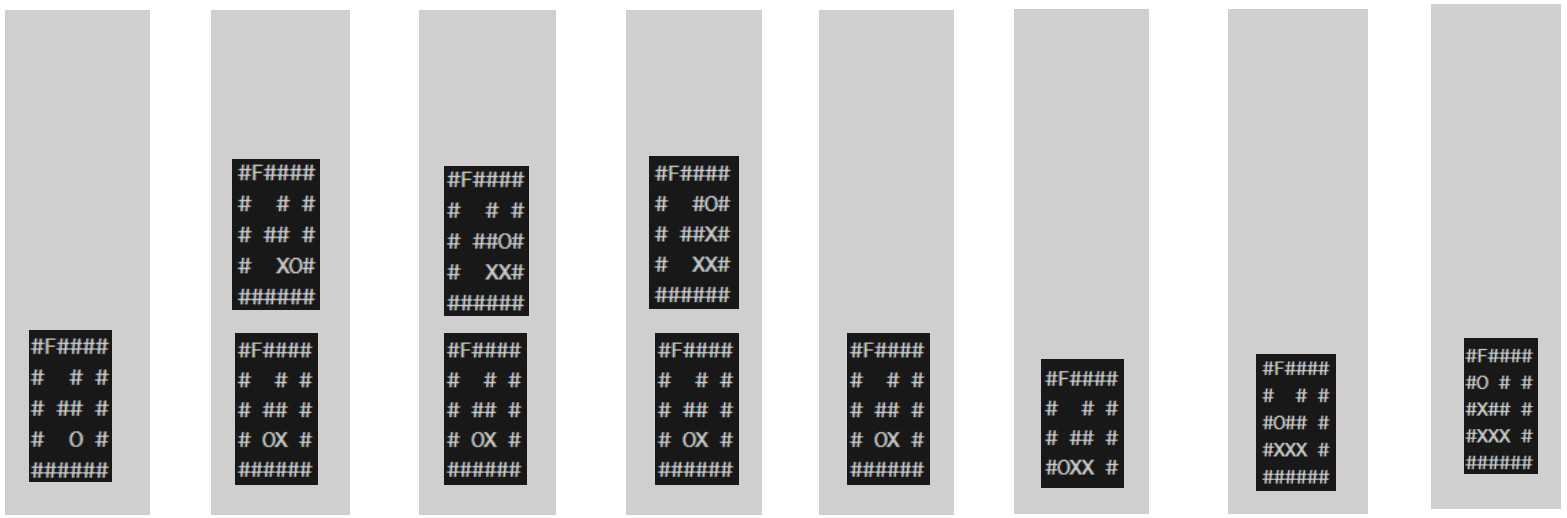
Example: State Space Search Trees

- A Labyrinth puzzle could also be solved using a state space search tree.



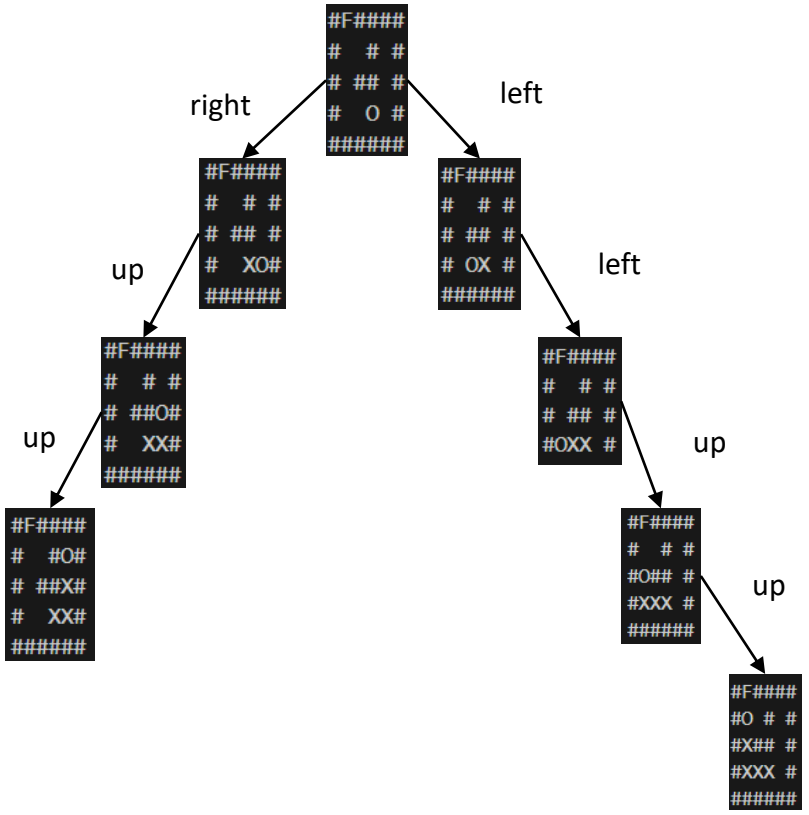
- The state space could be traversed through a stack or a queue.

Stack Solution:



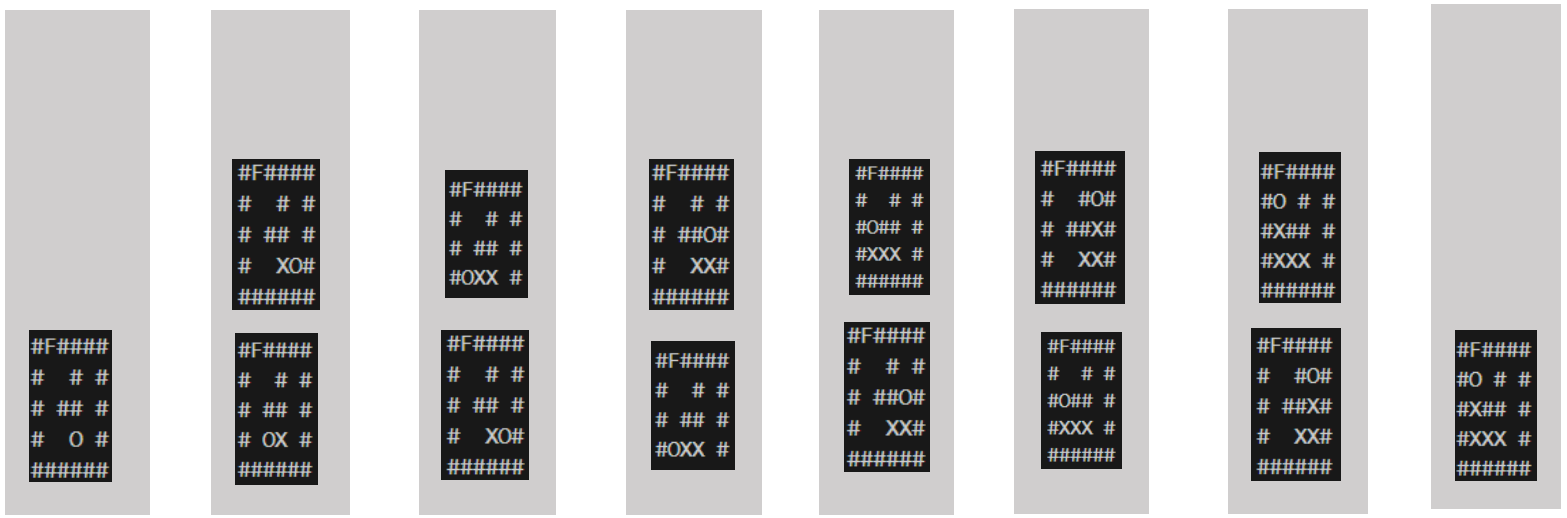
Example: State Space Search Trees

- A Labyrinth puzzle could also be solved using a state space search tree.



- The state space could be traversed through a stack or a queue.

Queue Solution:



BFS/DFS

- **Breadth First Search** starts at a source node and explores all nodes at the present depth level before moving on to the nodes at the next depth level. (Queue)
- **Depth First Search** starts at a source node and explores as far as possible along each branch before backtracking. (Stack)