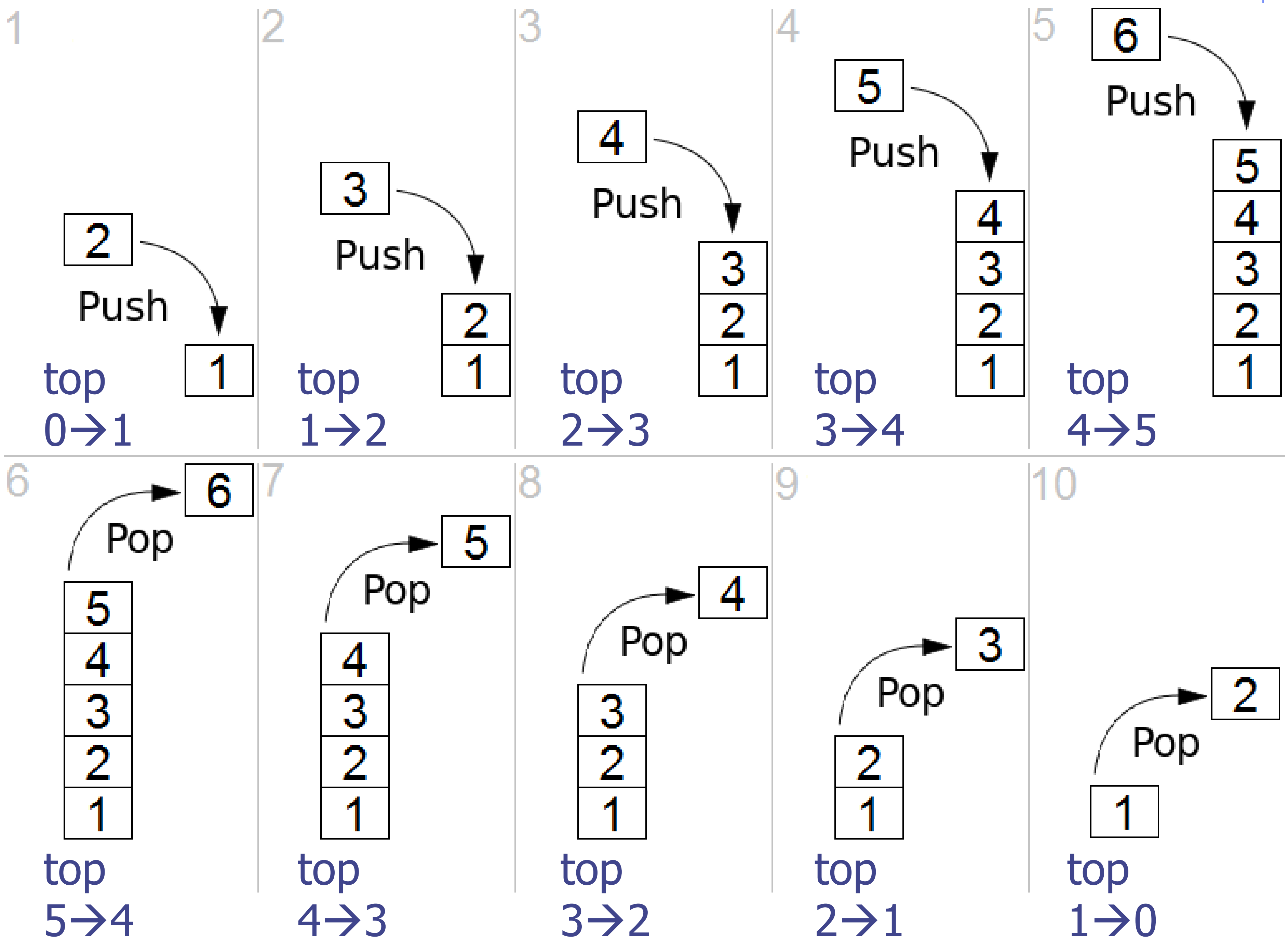
Lesson 9:
Stack and Queue

Jiun-Long Huang
National Chiao Tung University

Stack: Last-In-First-Out (LIFO) List

- Push
  - Add an element into a stack
- Pop
  - Get and delete an element from a stack


1
2
Push
1
top
0→1
2
3
Push
2
1
top
1→2
3
4
Push
3
2
1
top
2→3
4
5
Push
4
3
2
1
top
3→4
5
6
Push
5
4
3
2
1
top
4→5
6
6
Pop
5
4
3
2
1
top
5→4
7
5
Pop
4
3
2
1
top
4→3
8
4
Pop
3
2
1
top
3→2
9
3
Pop
2
1
top
2→1
10
2
Pop
1
top
1→0

Implementation of Stack by Array (contd.)

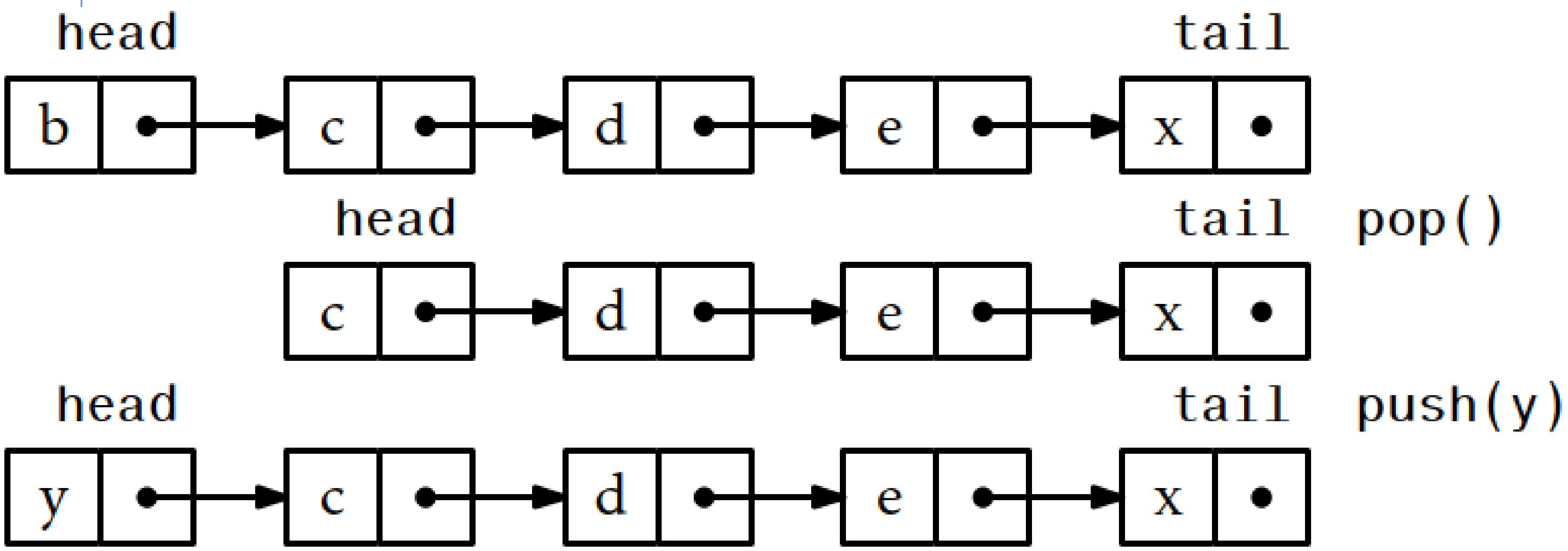
```
#define MaxSize 100
int top;
int stack[MaxSize];
void init(void){
  top=-1;
}
bool isFull(void) {
  if (top==MaxSize-1) return true;
  else return false;
}
bool isEmpty(void) {
  if (top==-1) return true;
  else return false;
}
```

Implementation of Stack by Array (contd.)

```
void push(int x) {
  /* add an item to the global stack */
  if (isFull()) printf("Stack is full");
  else stack[++top]=x;
}
int pop(void) {
  // return the top element from the stack
  int x;
  if (isEmpty()) {
    printf("Stack is empty");
    return 0;
  }
  x=stack[top--];
  return x;
}
```

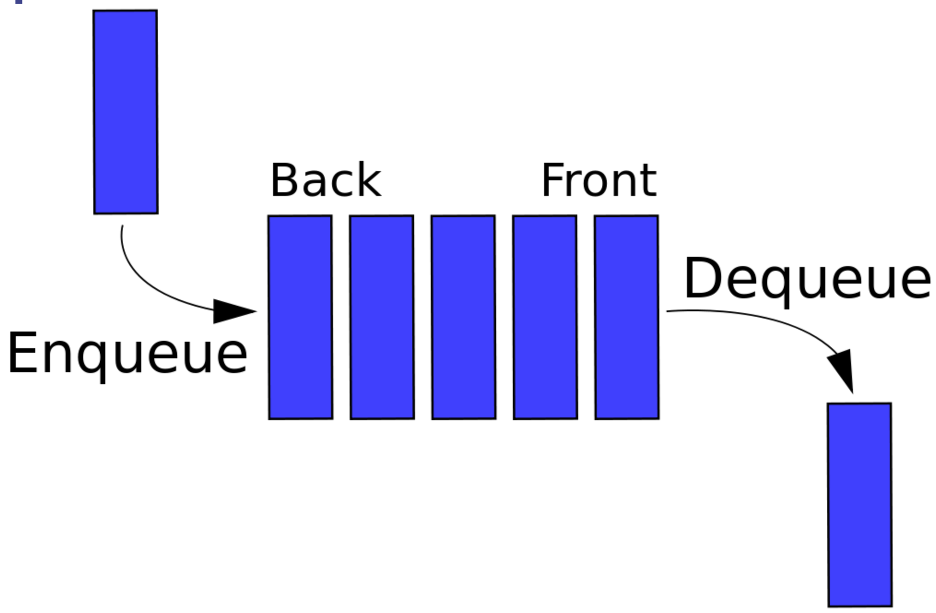
Implementation of Stack by Linked List

- Stack

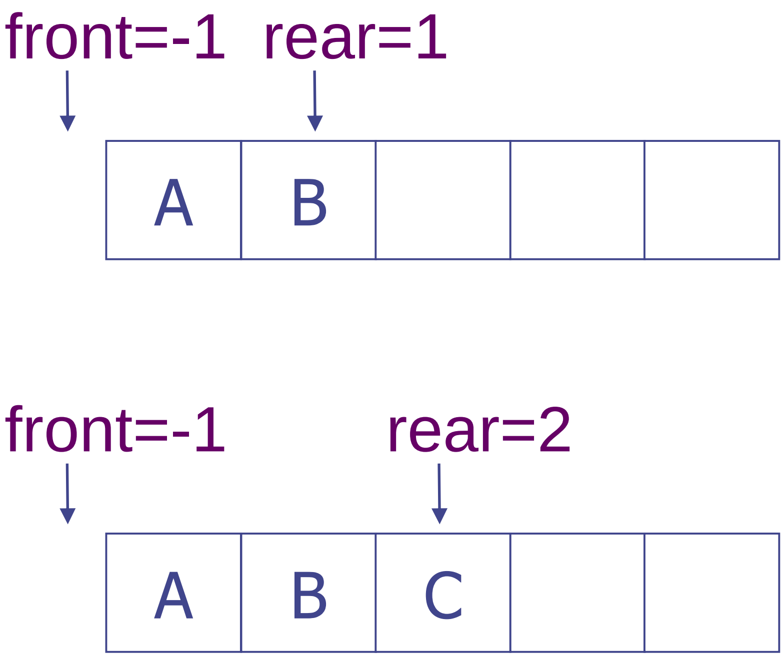


Queue: First-In-First-Out (FIFO) List

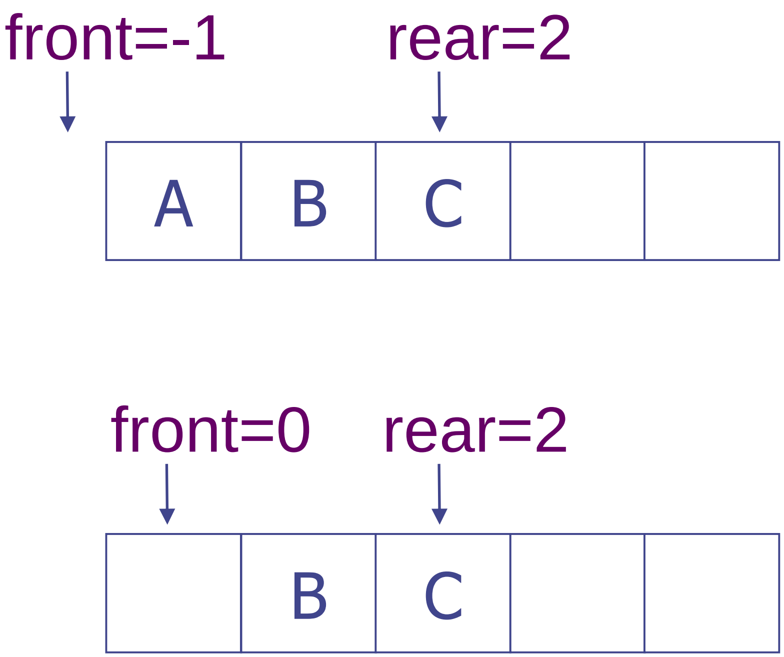
- Add an element into a queue
  - Get (enqueue) and delete (dequeue) an element from a queue
- Variation
  - Priority queue



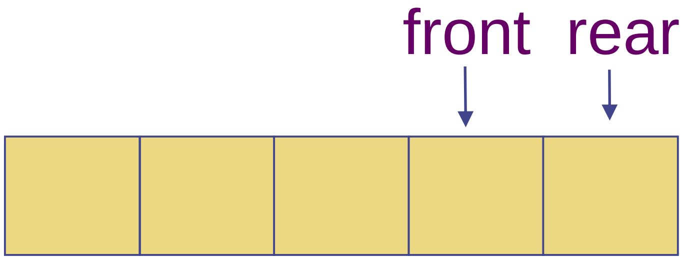
Enqueue



- Dequeue



Implementation of Queue by Array




```
#define MaxSize 100
int front,rear;
int queue[MaxSize];
void init(void) {
  front=rear= -1;
}
bool isFull(void) {
   if (rear==MaxSize-1) return true;
   else return false;
}
int isEmpty(void) {
   if (front==rear) return true;
   else return false;
}
```

Implementation of Queue by Array (contd.)

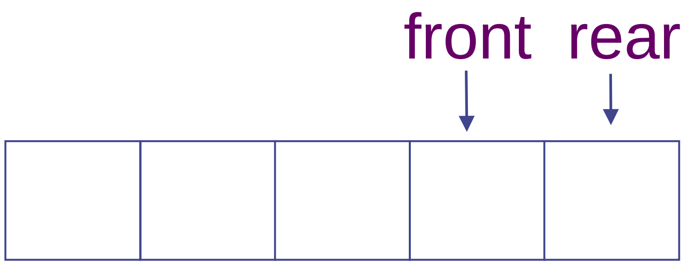
```
void enqueue(int x)
{
  /* add an item to the global queue */
  if (isFull())
    printf("Queue is full");
  else
    queue[++rear]=x;
}
```

Implementation of Queue by Array (contd.)

```
int dequeue(void)
{
  // return the front element from the queue
  if (isEmpty()) {
    printf("Queue is empty");
    return 0;
  }
  x=queue[++front];
  return x;
}
```

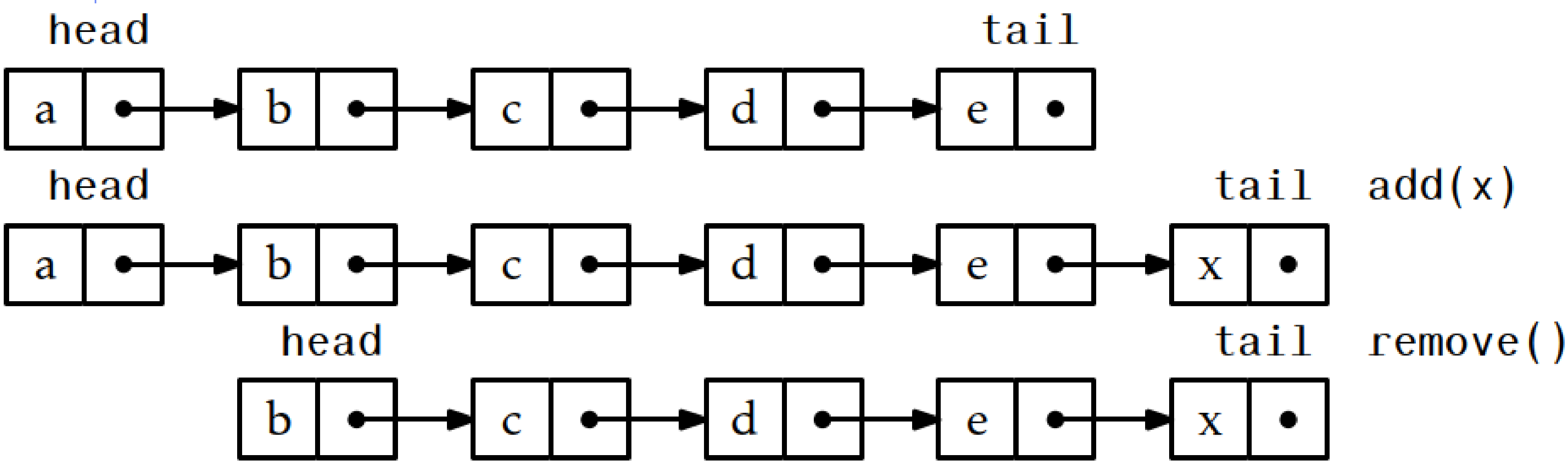
Problem

- As the elements enter and leave the queue, the queue gradually shifts to the right.
  - Eventually the rear index equals *MaxSize-1*, suggesting that the queue is full even though the underlying array is not full
- Solution:
  - Use a function to move the entire queue to the left so that *front=-1*
  - It is time-consuming
  - Time complexity=O(MaxSize)



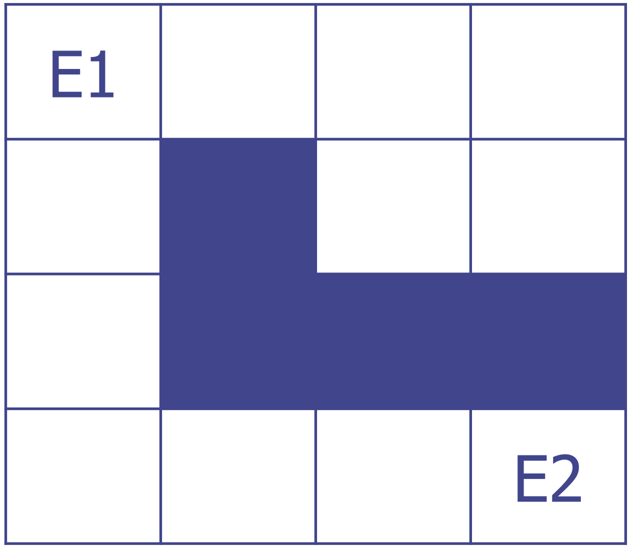
Implementation 2: By Linked List

- Queue



The Maze Problem

- Use two-dimensional array to model the maze



E1: Entrance
E2: Exit

```
int maze[][]={{0,0,0,0},
              {0,1,0,0},
              {0,1,1,1},
              {0,0,0,0}}
```

```
int offset_row[DIR_NO]={-1,0,1, 0};
int offset_col[DIR_NO]={ 0,1,0,-1};
```

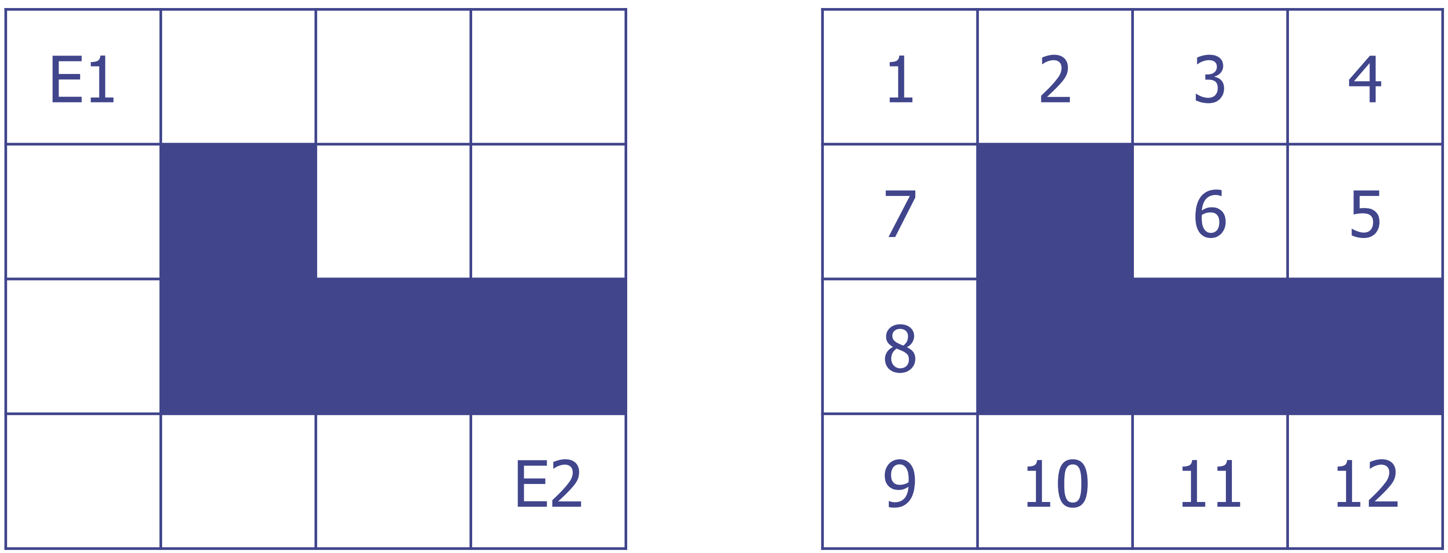
	row-1, col	
row, col-1	row, col	row, col+1
	row+1, col	

Solutions

- Depth-First Search
  - Use a stack
  - Use recursion
- Breadth-First Search
  - Use a queue

DFS: Depth-First Search

- Order of visit



Moving Forward

row=0, col=0

row=0, col=1

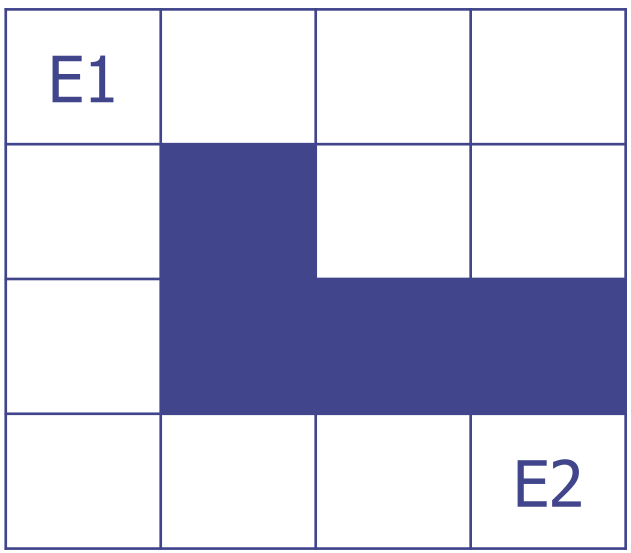
(0,0,1)

row=0, col=2

(0,1,1)
(0,0,1)

row=0, col=3

(0,2,1)
(0,1,1)
(0,0,1)



Backtracking

row=1, col=2

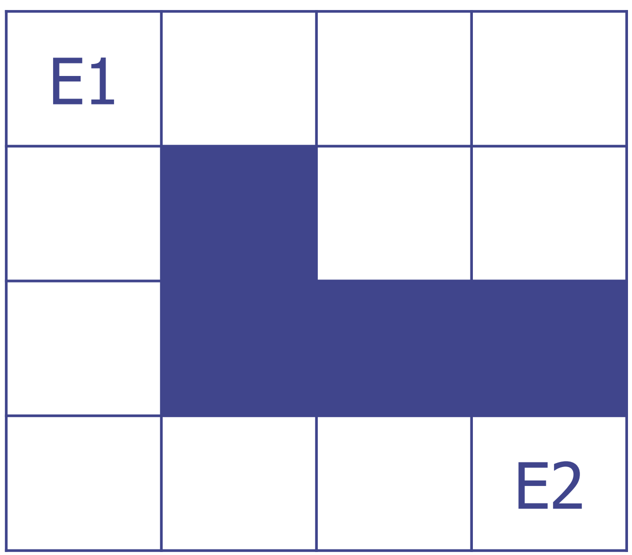
(1,3,3)
(0,3,2)
(0,2,1)
(0,1,1)
(0,0,1)

row=1, col=3

(0,3,2)
(0,2,1)
(0,1,1)
(0,0,1)

row=0, col=3

(0,2,1)
(0,1,1)
(0,0,1)



Backtracking

row=0, col=0

row=1, col=0

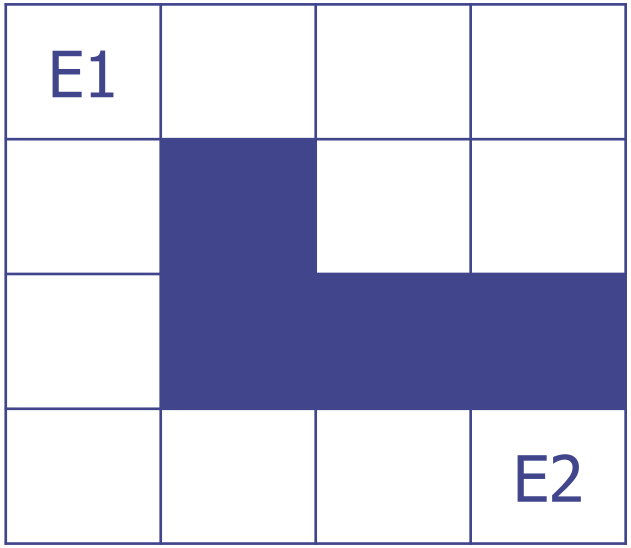
(0,0,2)

row=2, col=0

(1,0,2)
(0,0,2)

row=3, col=0

(1,2,2)
(1,0,2)
(0,0,2)



```
include <stdbool.h>
#include <stdio.h>
#define MAZE_WIDTH 4
#define MAZE_HEIGHT 4
#define DIR_NO 4
#define EXIT_ROW 3
#define EXIT_COL 3

int maze[MAZE_HEIGHT][MAZE_WIDTH]={{0,0,0,0},
                                   {0,1,0,0},
                                   {0,1,1,1},
                                   {0,0,0,0}};

int visited[MAZE_HEIGHT][MAZE_WIDTH]={0};
bool done=false;

int offset_row[DIR_NO]={-1,0,1, 0};
int offset_col[DIR_NO]={ 0,1,0,-1};
```

```
struct step
{
  int row, col, dir;
};
#define MaxSize 100
int top;
struct step stack[MaxSize];
void init(void){
  top=-1;
}
bool isFull(void) {
  if (top==MaxSize-1) return true;
  else return false;
}
bool isEmpty(void) {
  if (top==-1) return true;
  else return false;
}
```

```c
void push(struct step x) {
  if (isFull())
  printf("Stack is full");
  else
  stack[++top]=x;
}
struct step pop(void) {
  struct step x;
  if (isEmpty()) {
    printf("Stack is empty");
    return x;
  }
  x=stack[top--];
  return x;
}
```

```
bool isMovable(int row, int col)
{
  if (row>=0 && row<MAZE_HEIGHT && col>=0 &&
      col<MAZE_WIDTH && maze[row][col]==0 &&
      visited[row][col]==0)
    return true;
  else
    return false;
}

void dfs(int row, int col)
{
  int next_row, next_col,dir=0;
  int i;
  struct step sp;
  static int step_no;
```

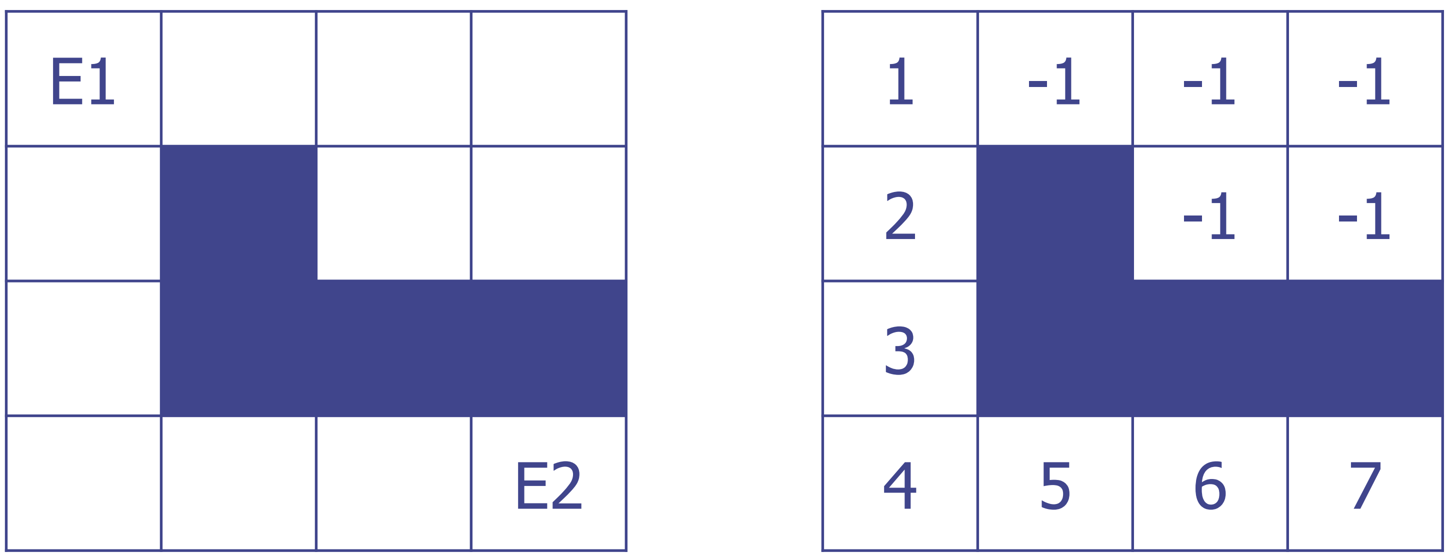
```
if (isMovable(row,col))
  visited[row][col]=++step_no;
else
  return;
while(true)
{
  if (row==EXIT_ROW && col==EXIT_COL)
  {
    done=true;
    return;
  }
```

```
for(i=dir;i<DIR_NO;i++)
{
  next_row=row+offset_row[i];
  next_col=col+offset_col[i];
  if (isMovable(next_row, next_col))
  {
    sp.row=row;
    sp.col=col;
    sp.dir=i;
    push(sp);
    row=next_row;
    col=next_col;
    dir=0;
       visited[next_row][next_col]=++step_no;
    break;
  }
}
```

```
    if (i!=DIR_NO)
        continue;
        do { // backtrack
          if (done==false)
            visited[row][col]=-1;
          step_no--;
          if (isEmpty()==false)
            sp=pop();
          else
            return;
          row=sp.row;
          col=sp.col;
          dir=sp.dir+1;
        } while (dir==DIR_NO);
    }
}
```

```
int main(void) {
  init()
  dfs(0,0);
  return 0;
}
```

Result



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

#define MAZE_WIDTH 4
#define MAZE_HEIGHT 4
#define DIR_NO 4
#define EXIT_ROW 3
#define EXIT_COL 3

int maze[MAZE_HEIGHT][MAZE_WIDTH]={{0,0,0,0},
                                   {0,1,0,0},
                                   {0,1,1,1},
                                   {0,0,0,0}};

int visited[MAZE_HEIGHT][MAZE_WIDTH]={0};
bool done=false;
int offset_row[DIR_NO]={-1,0,1, 0};
int offset_col[DIR_NO]={ 0,1,0,-1};
```

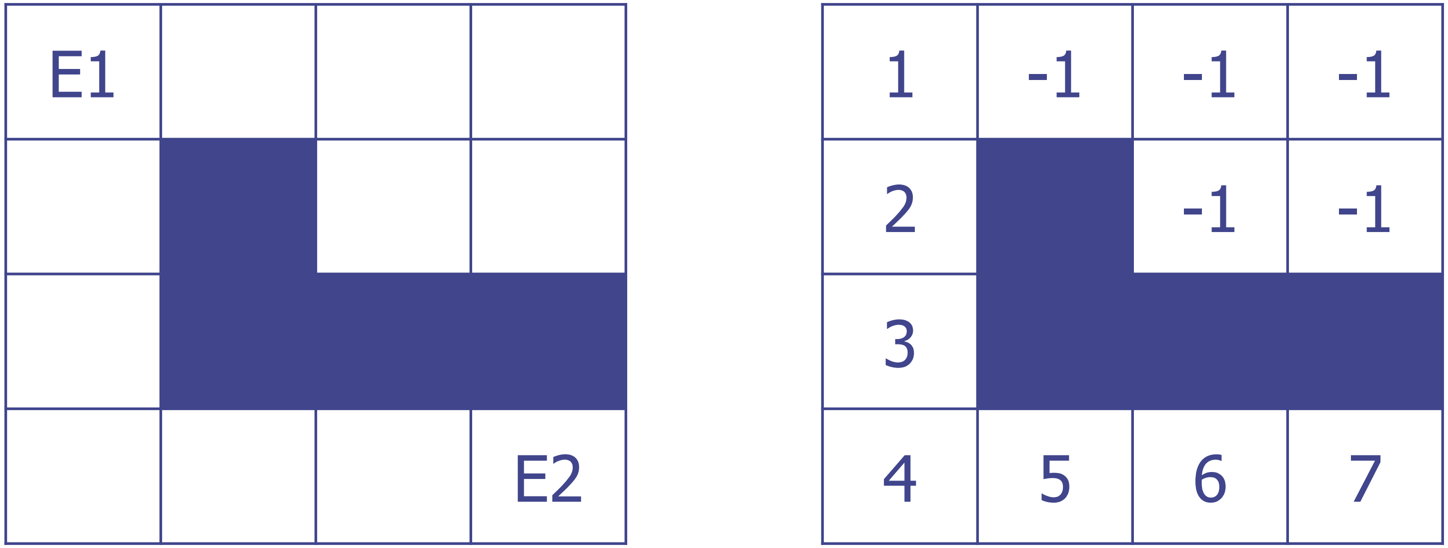
```
bool isMovable(int row, int col)
{
  if (row>=0 && row<MAZE_HEIGHT &&
      col>=0 && col<MAZE_WIDTH &&
      maze[row][col]==0 &&
      visited[row][col]==0)
    return true;
  else
    return false;
}
```

```
void dfs(int row, int col)
{
  int next_row, next_col;
  static int step_no;
  if (isMovable(row,col))
    visited[row][col]=++step_no;
  else
    return;
  if (row==EXIT_ROW && col==EXIT_COL)
  {
    done=true;
    return;
  }
```

```
 for(int i=0;i<DIR_NO;i++)
  {
    next_row=row+offset_row[i];
    next_col=col+offset_col[i];
    if (isMovable(next_row, next_col))
      dfs(next_row,next_col);
  }
  if (done==false)
    visited[row][col]=-1;
  step_no--;
}

int main(void) {
  init();
  dfs(0,0);
  return 0;
}
```

Result



Call Stack

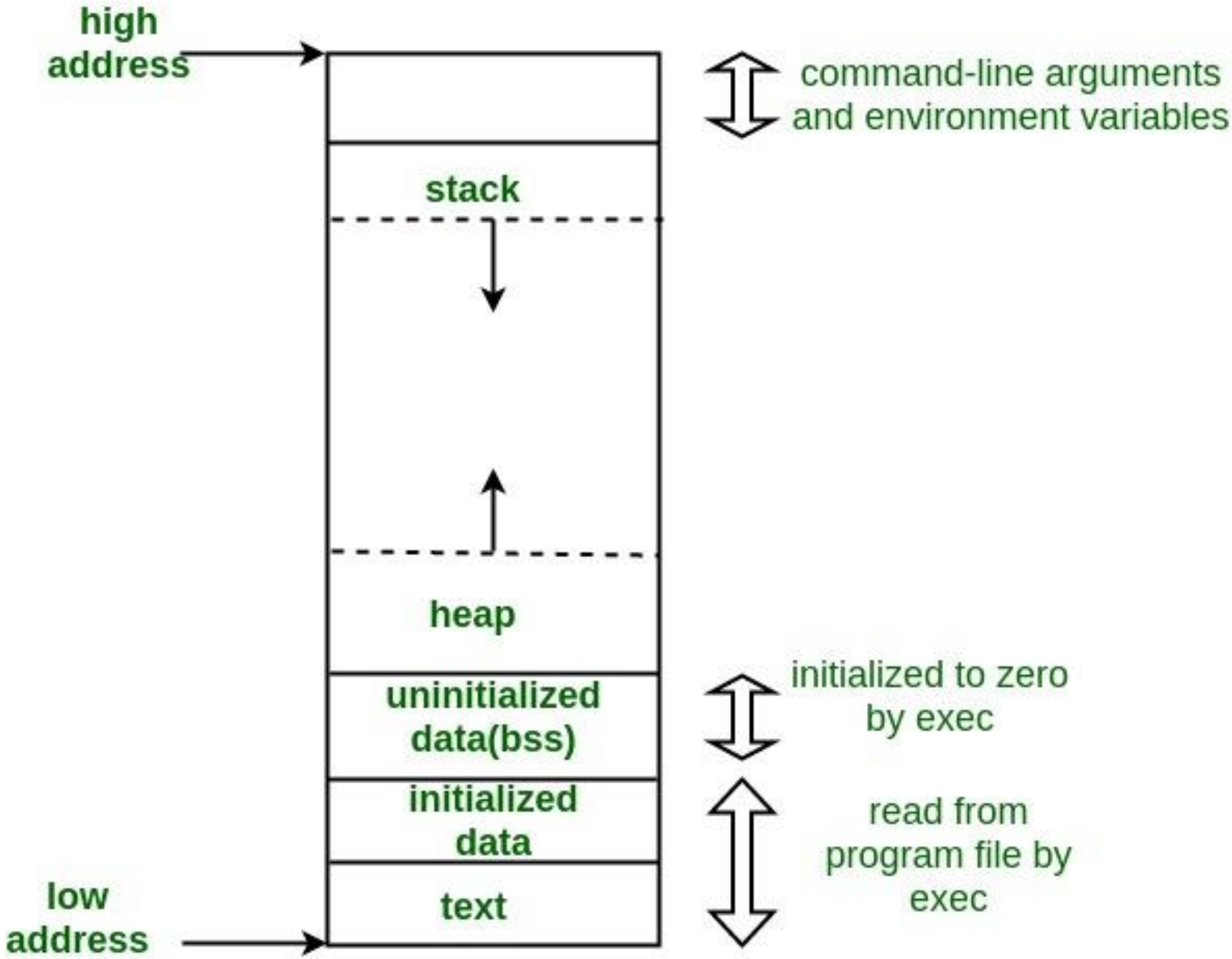
- A call stack is composed of stack frames (also called activation records or activation frames).
- Suppose that DrawLine() calls DrawSquare()

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

Memory Layout of C Programs

- A typical memory representation of C program consists of following sections.
  - Text segment
  - Initialized data segment
  - Uninitialized data segment
  - Stack
  - Heap

https://www.geeksforgeeks.org/memory-layout-of-c-program/



high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
low address
text

Text Segment

- A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.
- As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.
- Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Initialized Data Segment

- A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.
- Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

Uninitialized Data Segment (BSS Segment)

- Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing.
- Uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

Stack

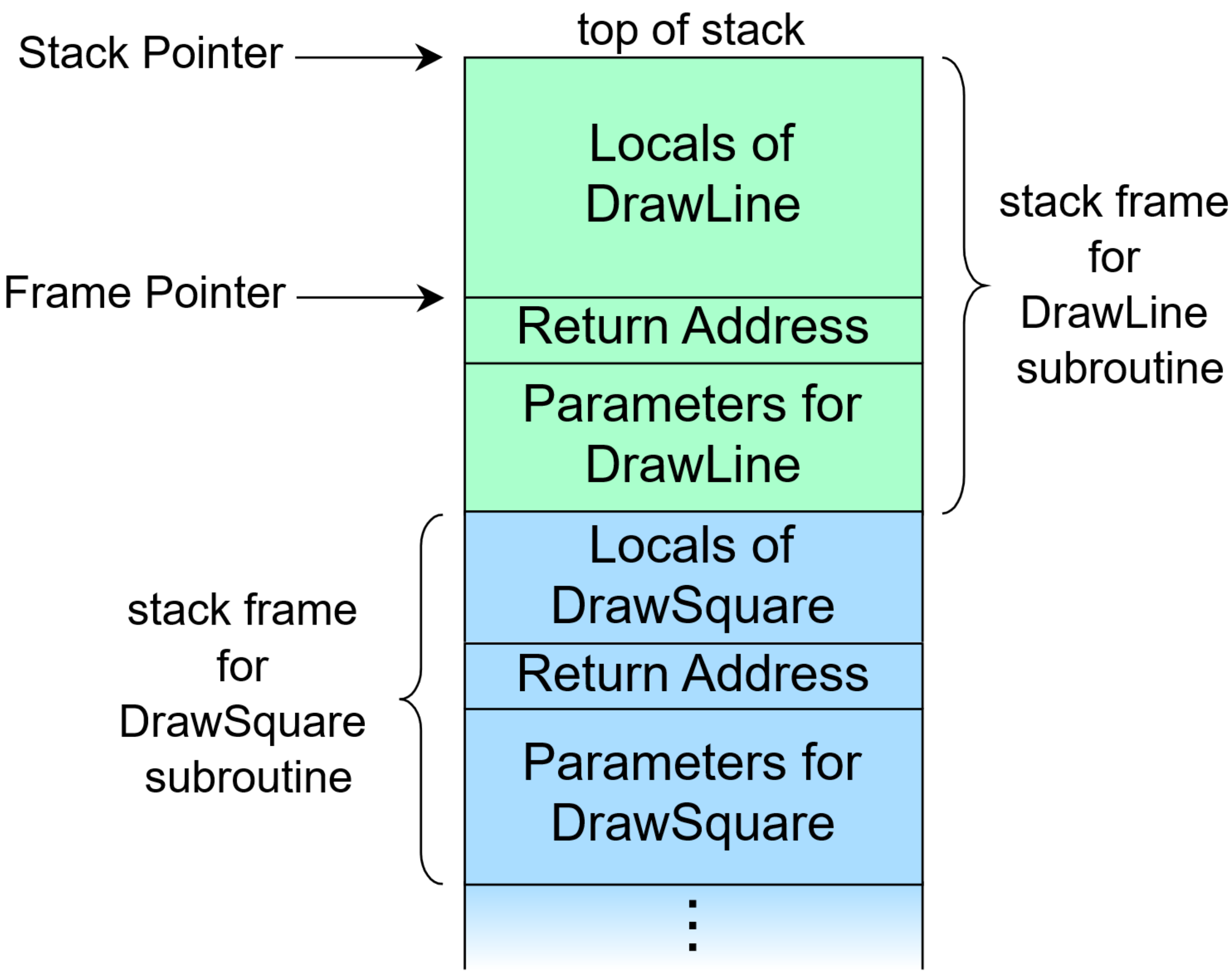
- The stack area contains the program stack.
- The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.
- Stack, where automatic variables are stored, along with information that is saved each time a function is called.

- Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack.

- The newly called function then allocates room on the stack for its automatic and temporary variables.
  - This is how recursive functions in C can work.
- Default size of program stack is usually small
  - Ex. 8MB in Linux

Heap

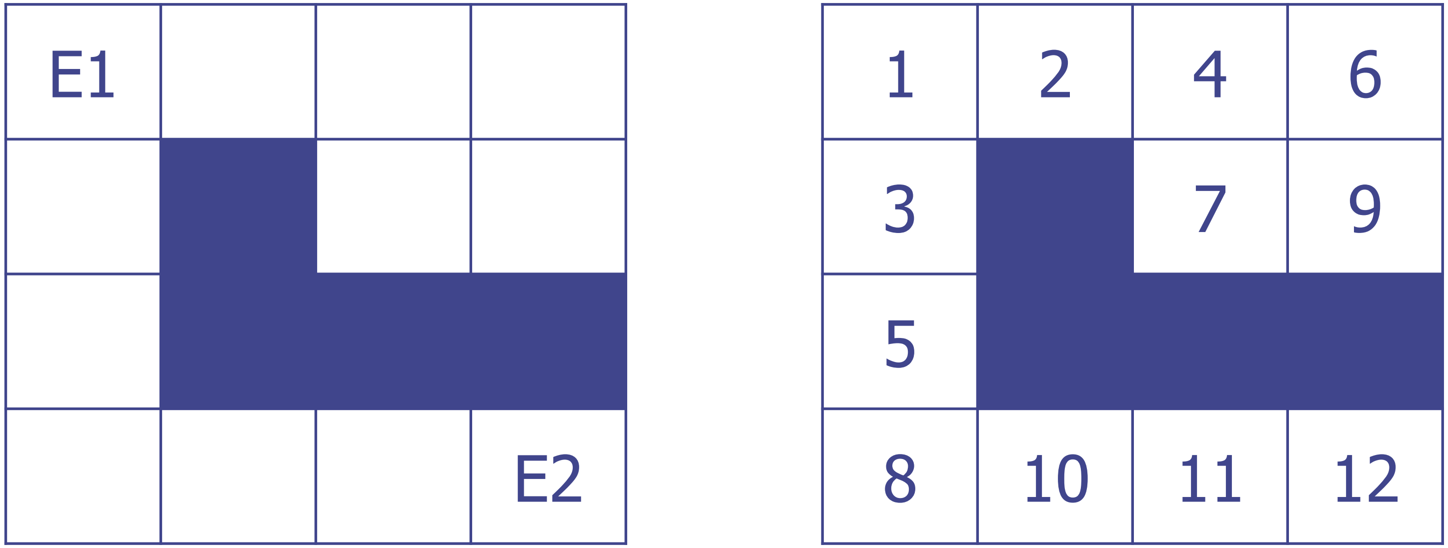
- Heap is the segment where dynamic memory allocation usually takes place.



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

BFS: Breadth-First Search

- Order of visit



Moving Forward

(0,0,1)

row=0, col=0

(0,1,2) (1,0,2)

row=0, col=1

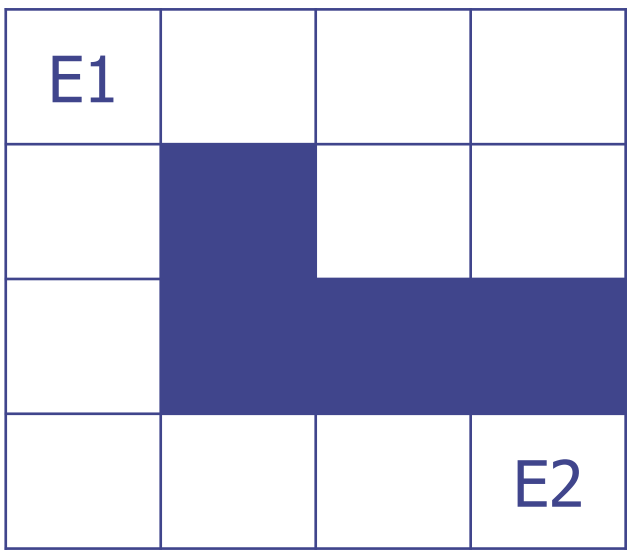
(1,0,2) (0,2,3)

row=1, col=0

(0,2,3) (2,0,3)

row=0, col=2

(2,0,3) (0,3,4) (1,2,4)



```c
#include <stdbool.h>
#include <stdio.h>
#define MAZE_WIDTH 4
#define MAZE_HEIGHT 4
#define DIR_NO 4
#define EXIT_ROW 3
#define EXIT_COL 3

int maze[MAZE_HEIGHT][MAZE_WIDTH]={{0,0,0,0},
                                   {0,1,0,0},
                                   {0,1,1,1},
                                   {0,0,0,0}};

int visited[MAZE_HEIGHT][MAZE_WIDTH]={0};
bool done=false;

int offset_row[DIR_NO]={-1,0,1, 0};
int offset_col[DIR_NO]={ 0,1,0,-1};
```

```
struct step
{
  int row, col, step_no;
};
#define MaxSize 100
int front,rear;
struct step queue[MaxSize];
void init(void) {
  front=rear= -1;
}
bool isFull(void) {
  if (rear==MaxSize-1) return true;
  else return false;
}
int isEmpty(void) {
  if (front==rear) return true;
  else return false;
}
```

```
void enqueue(struct step x)
{
  if (isFull())
    printf("Queue is full");
  else
    queue[++rear]=x;
}
struct step dequeue(void)
{
  struct step x;
  if (isEmpty()) {
    printf("Queue is empty");
    return x;
  }
  x=queue[++front];
  return x;
}
```

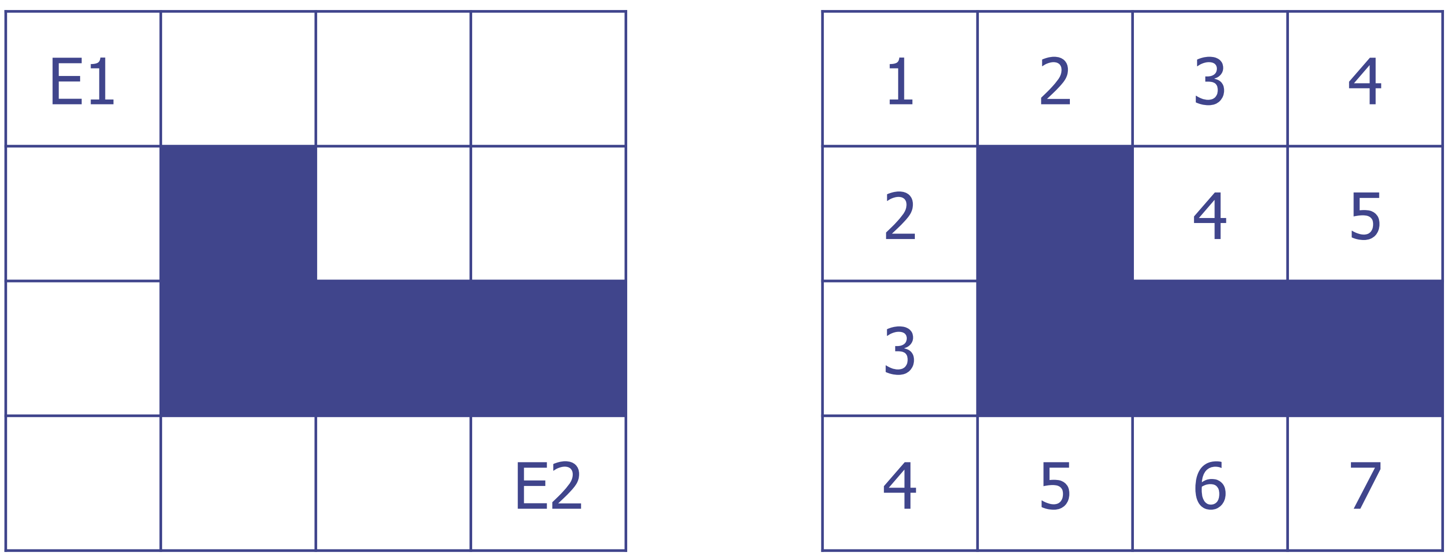
```
void bfs(int row, int col)
{
  int i, step_no, next_row, next_col,dir=0;
  struct step sp;
  if (isMovable(row,col))
  {
    sp.row=row;
    sp.col=col;
    sp.step_no=1;
    enqueue(sp);
  }
  else
    return;
```

```
while(isEmpty()==false)
{
  sp=dequeue();
  row=sp.row;
  col=sp.col;
  step_no=sp.step_no;
  visited[row][col]=step_no;
  if (row==EXIT_ROW && col==EXIT_COL)
  {
    done=true;
    return;
  }
```

```
    for(i=dir;i<DIR_NO;i++)
    {
      next_row=row+offset_row[i];
      next_col=col+offset_col[i];
      if (isMovable(next_row, next_col))
      {
        sp.row=next_row;
        sp.col=next_col;
        sp.step_no=step_no+1;
        enqueue(sp);
      }
    }
  }
}
```

```
int main(void) {
  init();
  bfs(0,0);
  return 0;
}
```

Result



Comparison

- Use BFS to solve the maze problem
  - Pros:
    - The first path obtained is of the shortest path
  - Cons:
    - High memory consumption

- Use DFS to solve the maze problem
  - Pros:
    - Low memory consumption
  - Cons:
    - DFS has to search all paths in order to obtain the shortest path

Reference

- Stack (abstract data type)
  - https://en.wikipedia.org/wiki/Stack_(abstract_data_type)
- Queue (abstract data type)
  - https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
- Pat Morin, Open Data Structures: An Introduction.
  - http://opendatastructures.org/ods-cpp/