Stack

- start of empty

• pus x

• push y

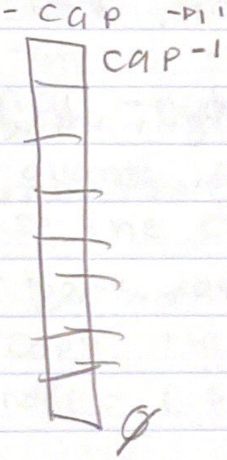
push z

z   z ◁— pop

y   y ◁— pop

x

- stack has items / arrays
- top -> int
- cap -> int (capacity)



top always points to the next available spot

-> top = 0, means it's emty

Recursion

- defined in terms of itself
  - ↳ use it with itself

example =
```
int sum_1(int k) {
    if (k>1) {
        return k + sum_1(k-1);
    }else {
        return 1;
    }
}
```

- Just because you can use recursion, it does not mean you should!

Tail recursion → iteration with right hand

function call requires creating a stack frame
  - ↳ takes time & space
- often time recursion can be written as iteration
- call a function by pushing
- static → not in a stack
- global variables → ~~not~~ only one copy
- use it when it makes code clearer

Binary Search

- search an ordered array in $O(\log(n))$
- If list == empty, then its not there
- If list == small, look in the left half
- If list == large, look in the right half

binary search is as fast as we can go
- binary search requires that the array be sorted

String Table

~~Stri~~ include <strings.h>

char * entry

struct str_tree * left, * right;

Str_tree; * root → node

→ String that stores only one copy

- look at old notes on binary trees

- function that finds & inserts

new_node()

- allocate a node

- se the children to null

- make room for the string

- copy the string

node = (str_tree *) malloc (size of (

Str-find()

- non-recursive
- Start at the root
- have a curser
- if no children == done

recursion is natural for search
↳ by dividing the space
- we calculate, search, and sort

top always points to the next available space

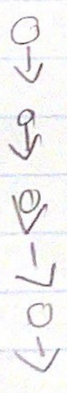
Out perameter of function return

Stack FS = ∅ (empty)

s → p o → next =

push(s)

$\left.\begin{array}{l} p = \text{malloc}() \\ p \to \text{next} = s \\ s = p \end{array}\right\}$ push



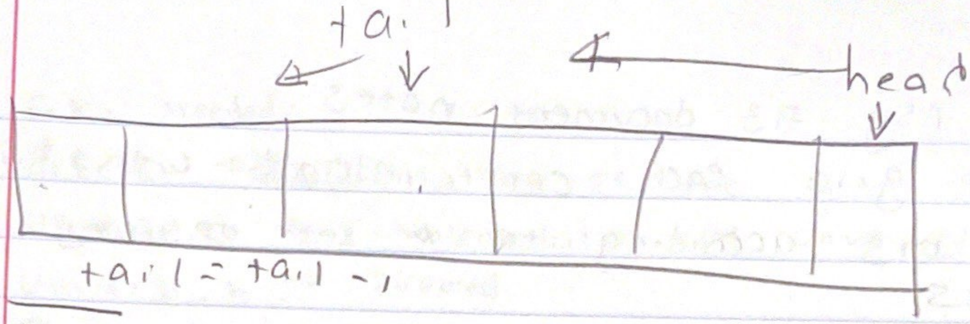
← top

pop:

$\left.\begin{array}{l} t = \text{top} \\ x = t \to \text{item} \\ \text{top} = t \to \text{next} \end{array}\right\}$ pop

free(t)

→ memory leak



heand = head - 1

If h & t = same space it's emty

to insert, you can s

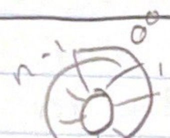
insert O(n)

insert m things

= O(n m)

~ O(n2)

Bad :(



insert

q[n] = x

n = n+1 % n

remace:

x = q[z]

z = z+1 % n

circular q - thing of left/right vamp

use order

Asg #3 document notes

- 2-D grid each cell interacts with its neighbors according to a set of rules

Tasks

- should be played in a infinite, two-D grid of cells that represent a universe
  - only two states exist: alive or dead
- game progresses through generations
  - rules for each generation:
    1.) any live cells with 2 or ~~m~~ 3 live neighbors survive
    2.) any dead cell with exactly 3 live neighbors becomes alive cell
    3.) all other cells died due to loneliness or overcrowding

The universe

- will be an abstract data type
  - finite 2-d grid of cells

typedef to construct a new type

false == dead

True == alive

valgrind = check for memory leaks

memory allacation

stack - grows down, ints/variable
- behind the scene

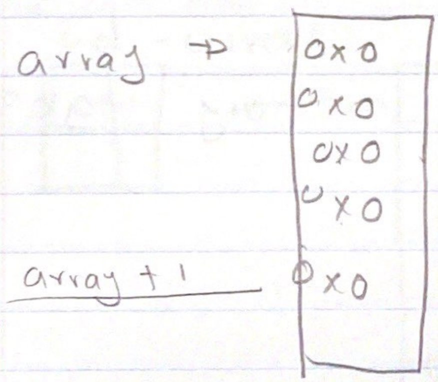
heap —> manually request memory

Two different walls to heap

1.) malloc() —> espects size as a paramenter

2.) calloc) —> return pointers
 —> spects 2 parameters #, items, how big is each item

malloc	calloc
- used more - int * array = (int *) calloc(100, sizeof(int));	returns a void ponter



array + 1 —> increment poin by size of int

1.) allocate space
2.) free

Struct

```
struct coor {
    int x;      (4)
    int y;      (4)
    char * str; (8)
}
```

push on the stack

```
struct coor a;
a.x = 1;
a.y = 2;
a.str = "hello"  x x x
char *c] = "hello"
```

```
struct * coor *a =
  (struct coor*) calloc
  (1, sizeof(struct coor))
a-> str = (char*)
  calloc(10, sizeof
  (char));
```

First in, last out

When you free memory, you do it backwards

~~free (arr);~~

```
free(c-> str);
free (c);
```

valgrind -> checks for memory leaks

• two universes:

asg # 3 include it here

w.h -ordered -> w.c <- life.c

Struct Universe implementation • use all

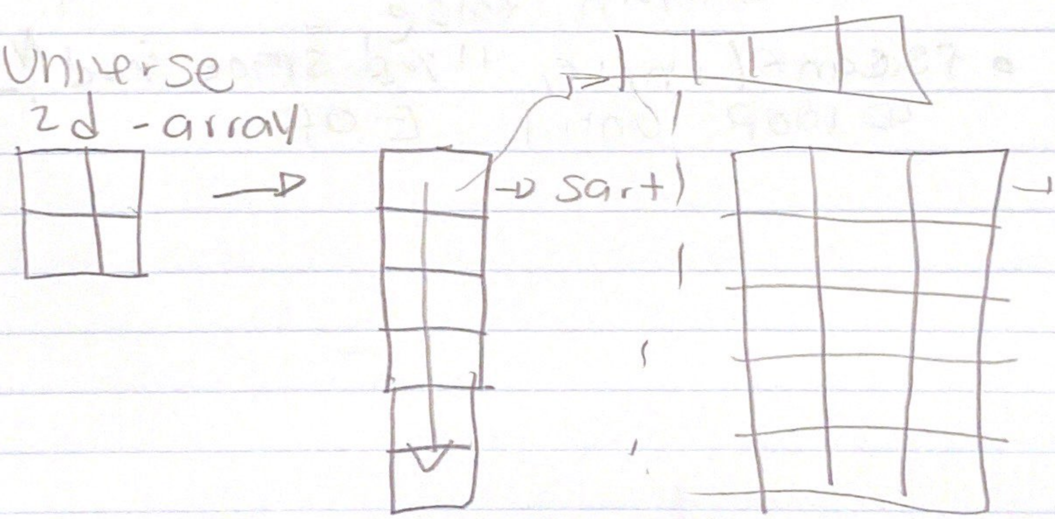
• def U (given) implementation

• implement f's

run valgrind --leak-check=full
--show-leak-kinds=all -s ./life -i lists/glider.txt

1.) universe —— row
| column
2.) populating
3.) getting a cell & neighbors

Universe
2d-array



first allocate row, then each row a column

36 = rows   65 = columns

U -> rows
U -> colums

if (U == null)
return 0
U -> = ____

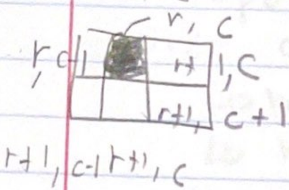
Alive true
define Dead false

get
return alive or dead

UV_ populate(universe, file)
- check for null
  -> return false
- fscanf(infile, "%d space %d"...)
  -> loop until EOF

→ looks at one cell

Consensusct → (v, r, c)



get-cell x 7

r ≥ round cr

r < 0, no neighbor
return false

(r + rows) % rows → out of bounds in cont world