Plan:

- Stacks
  - Impl
  - Use cases
  - Call stack
  - Algo: Infix → Postfix
  - Algo: DFS
  - Array based vs. Linkedlist based
- Queues
  - Impl
  - Circular
  - Use cases
  - Algo: BFS
- STL data structures
- Trees, Hashmap (Briefly)

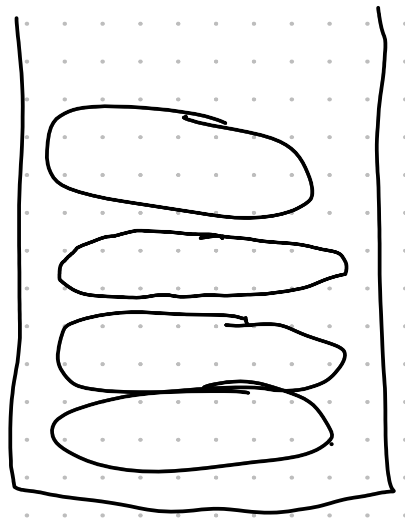
Logistical Announcements

- HW2 tonight (implement in the .h files)
  // TODO
- HW3 out tmr
- Project out tmr
- Exit diagnostic on Fri.
- Final Sat 6-8 pm

Stack

- "Abstract Data Type"

  - Only interact at the "top"

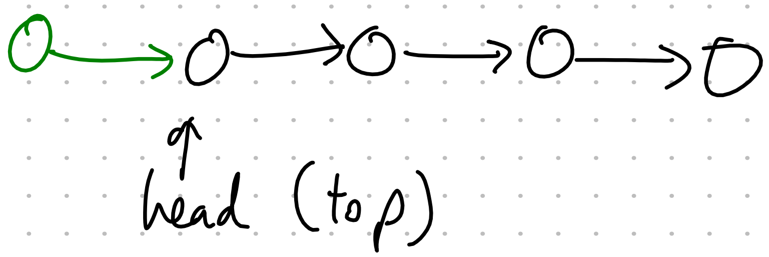


```
stack<int> s;
 s.push(5);
 s.pop();
cout << s.top()
```

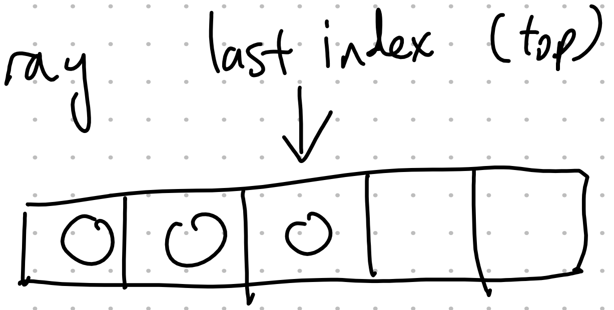
```
class Stack {

   methods()...

}
```

Impl:

- Linked list



- Array



5 → 10 → 20 → 40

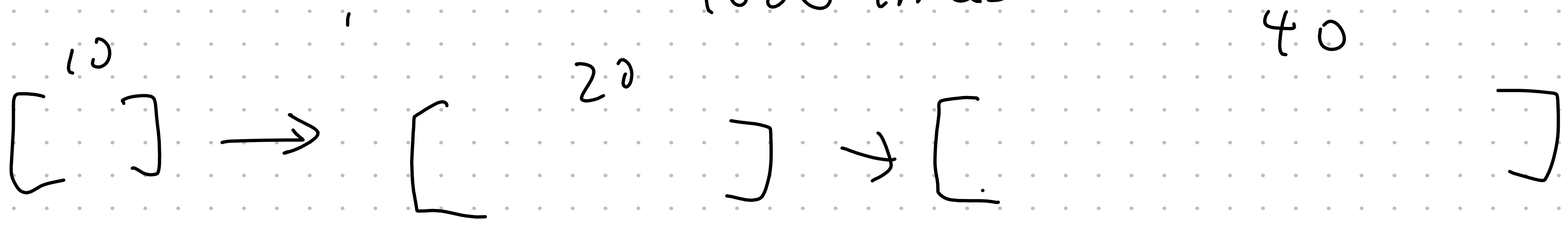
"Amortized cost"

Vector: Exactly what we just described

v.push_back(1)

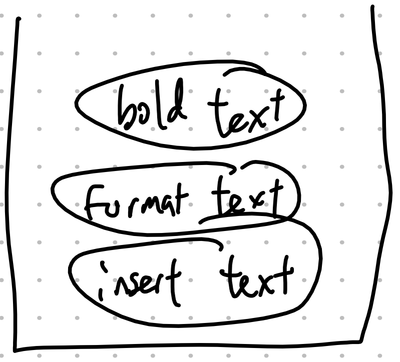
⋮

1000 times



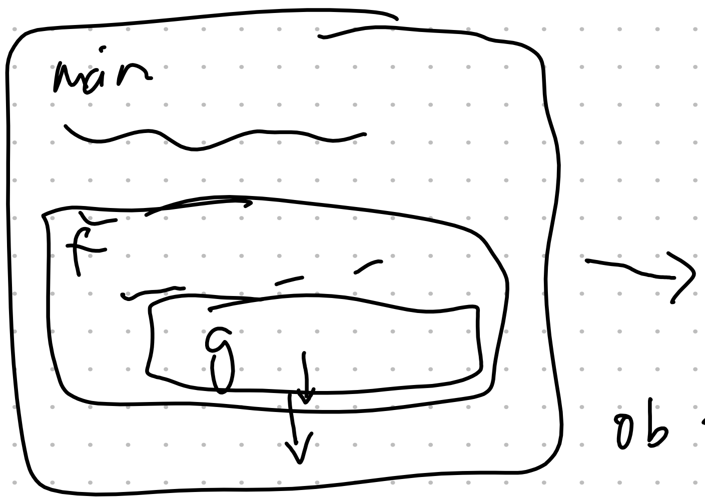
Stack Use Cases

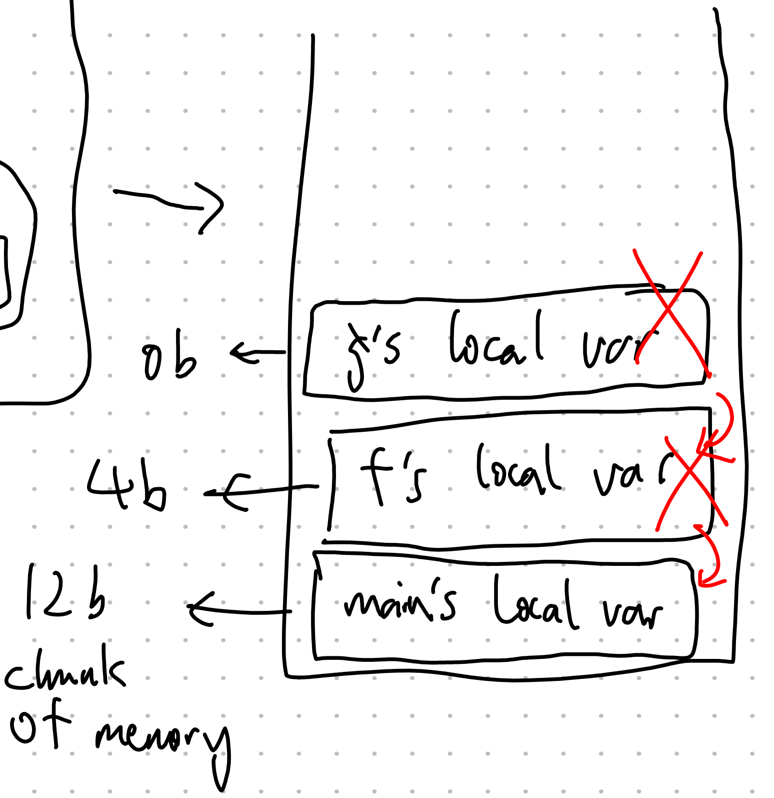
- Undo / redo
- Dispenser
- Call frames



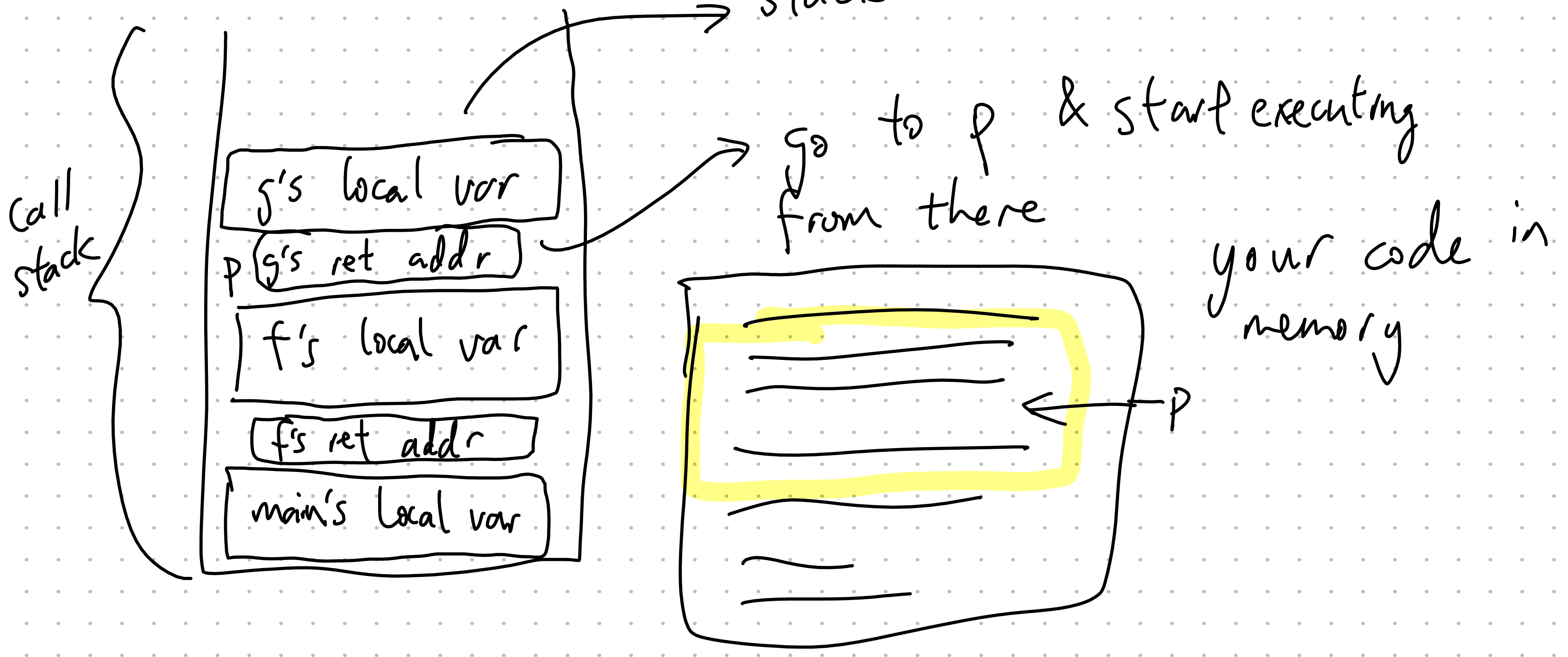
```
    int main {
4b      int x;
8b      long long y;
        f()
    }
```

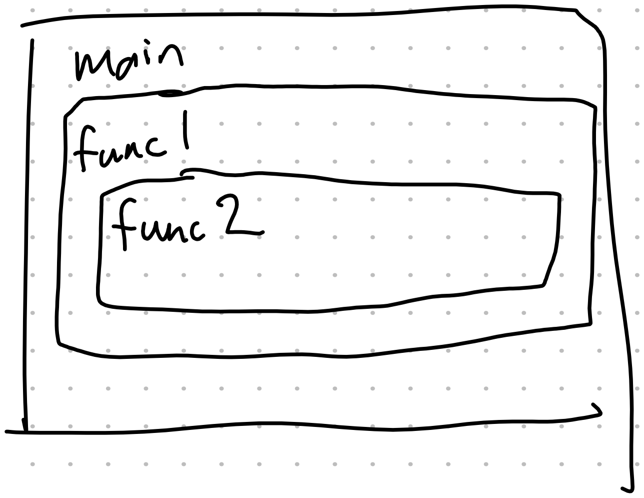
```
    void f() {
4b      int z;
        g()

        cout << z
    }
```

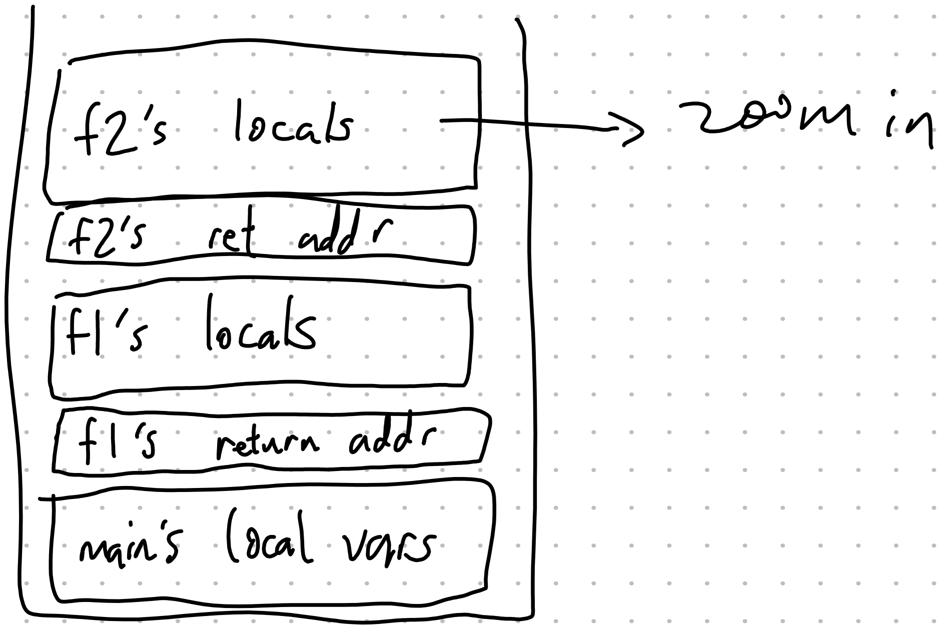


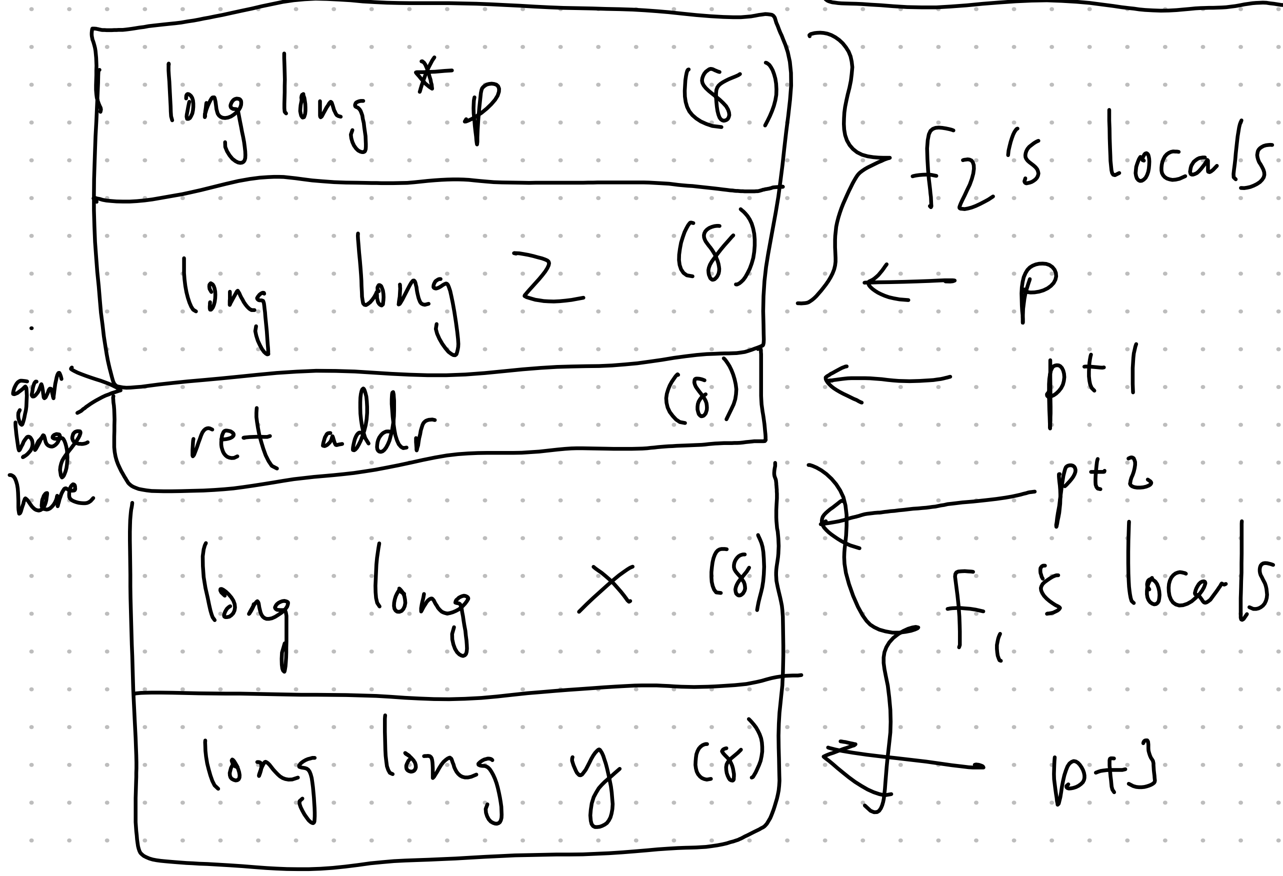


Return address (noun)




main
func1
func2

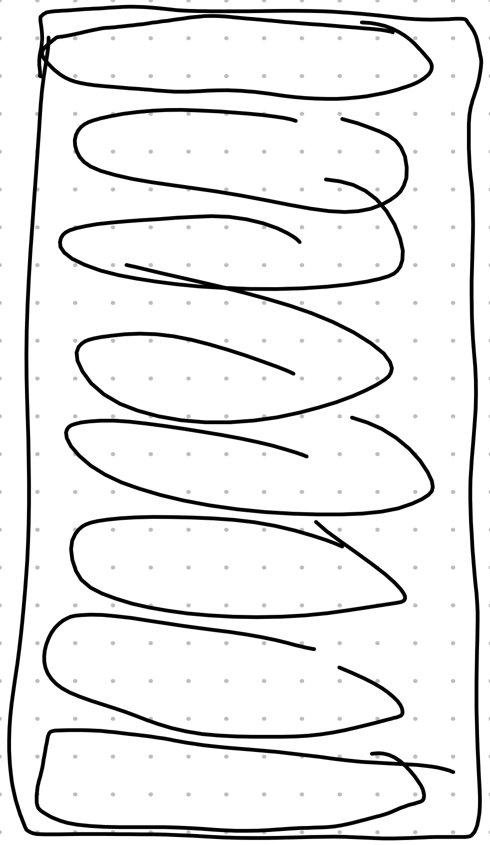

f2's locals
zoom in
f2's ret addr
f1's locals
f1's return addr
main's local vars

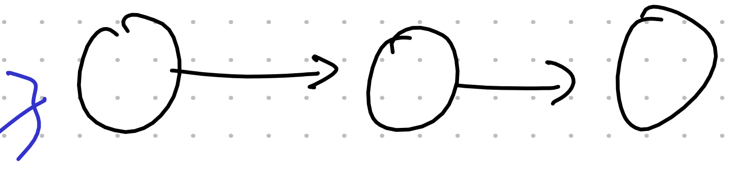

long long *p (8)
long long z (8)
ret addr (8)
long long x (8)
long long y (8)
f2's locals
f1's locals
p
p+1
p+2
p+3
garbage here

*(p+3)

p[3]

Call stacks are array based





linked list

Go (golang)

- stack frames are allocated on the heap (dynamically allocated)

Algo: Infix → Postfix

PEMDAS

"$(a+b \times c) \times d$" infix

"$b\ c \times a + d \times$"

$b \times c = x$

$x + a = y$

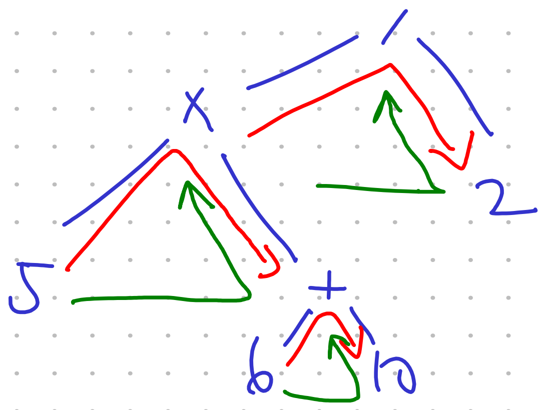
$y \times d =$ final ans

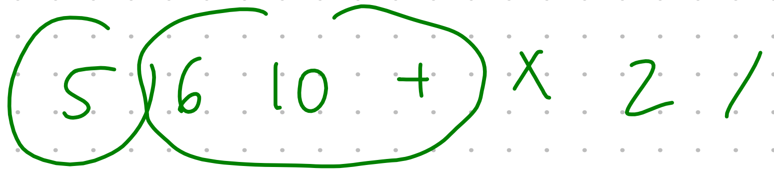
Infix to Postfix Conversion

Inputs: infix string
Output: postfix string (initially empty)
Private data: a stack

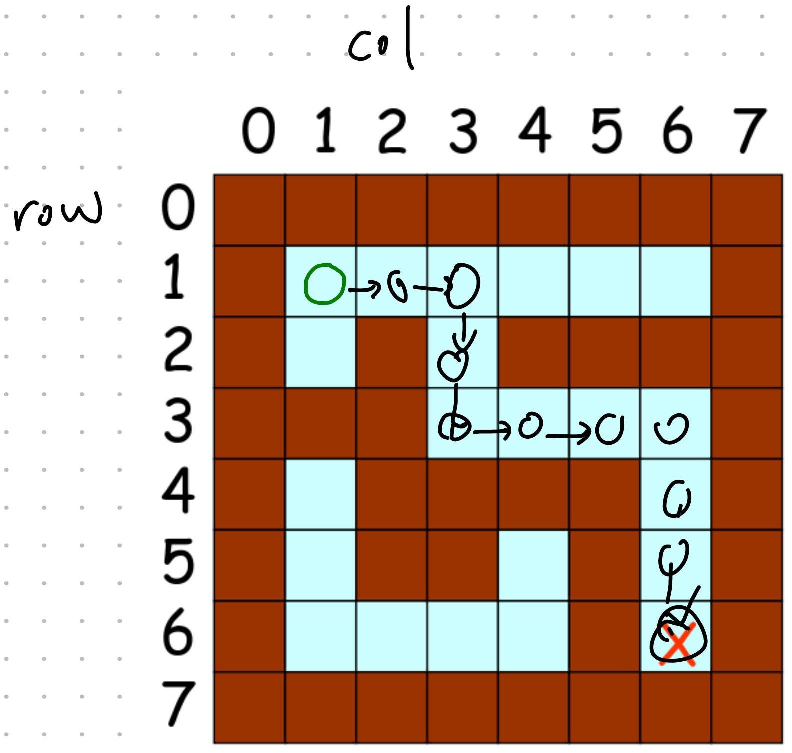
1. Begin at left-most infix token.
2. If it's a #, append it to end of postfix string followed by a space
3. If its a "(", push it onto the stack.
4. If it's an operator *and the stack is empty*:
   a. Push the operator on the stack.
5. If it's an operator and the stack is NOT empty:
   a. Pop all operators with greater or equal precedence off the stack and append them on the postfix string.
   b. Stop when you reach an operator with lower precedence or a (.
   c. Push the new operator on the stack.
6. If you encounter a ")", pop operators off the stack and append them onto the postfix string until you pop a matching "(".
7. Advance to next token and GOTO #2
8. When all infix tokens are gone, pop each operator and append it } to the postfix string.

$5 \cdot (6+10)/2$ $\quad 5 \cdot \frac{6+10}{2} \quad \frac{5 \cdot (6+10)}{2}$





We are traversing the tree using Depth First Search algo: DFS



solve (grid, start_row, start_col
end_row, end-col)

stack<coord> s;
struct coord { int r; int c; };

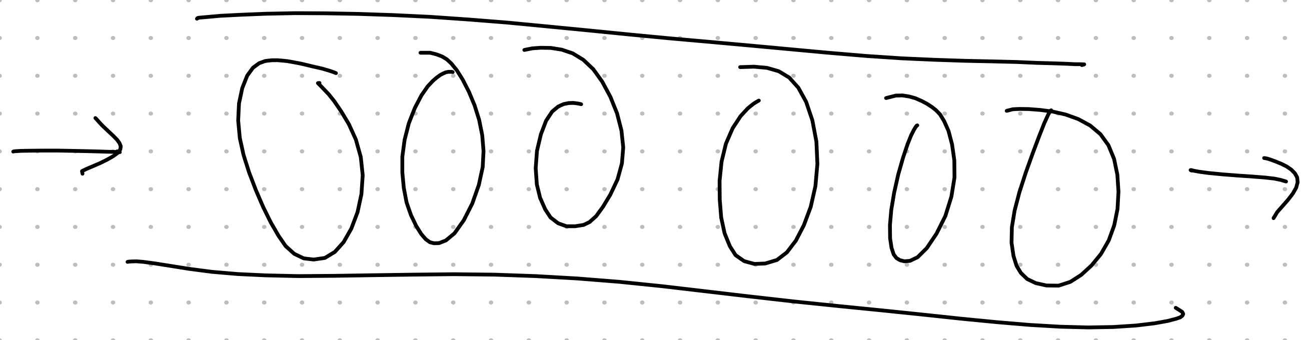
1. check if we're at X

2. check top, down, left, right (except for direction we came from) for available cells

3. go to available cell
if stuck, pop the stack and go to prev. cell

4. if checked all neighbors, and none worked,

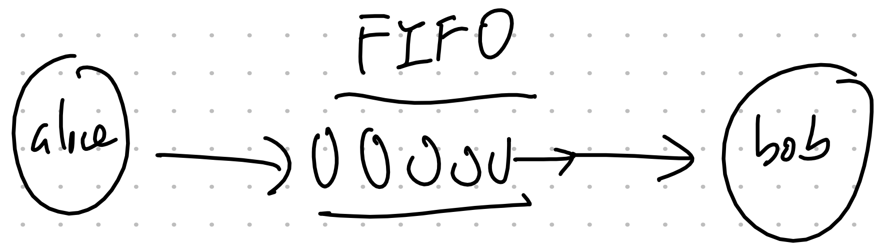
Queues

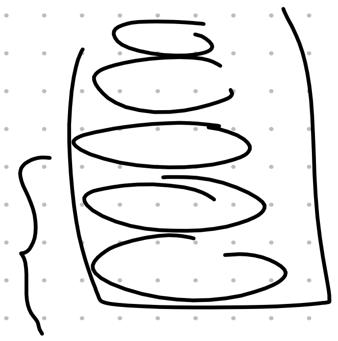


- Linked list
- Array (dyn resize or circular)

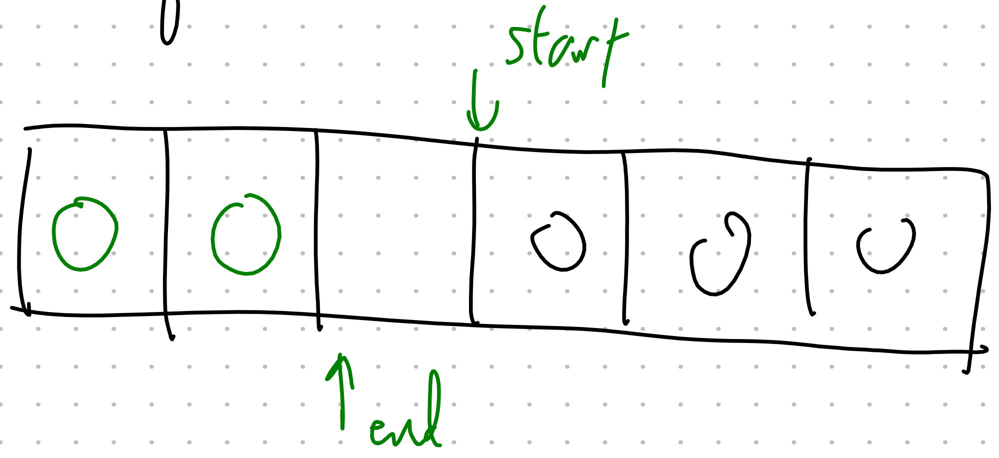
Use cases

- Message passing





Circular queue



push() x4

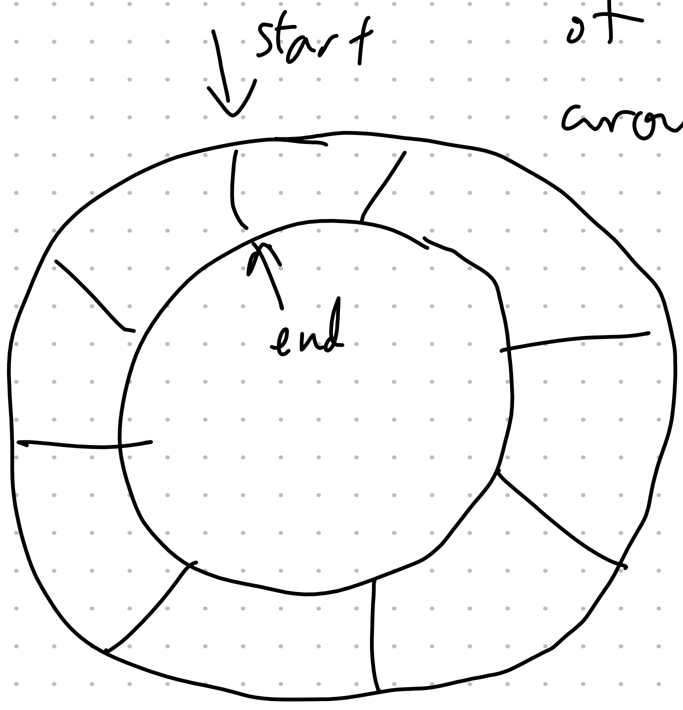
pop()

push() x2

push()

pop() x2

push()



- Fixed size (so can't push more than 6 at a time)
- Pops free space in the queue so we can make use of it by wrapping around

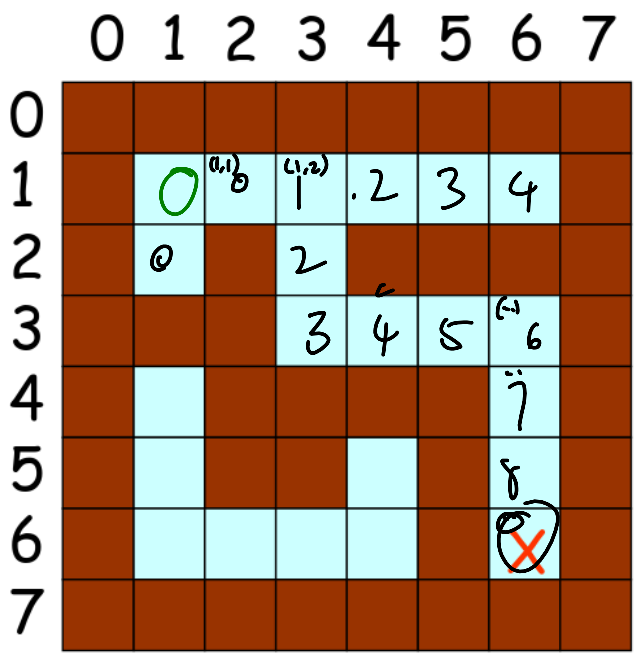
Algo: BFS

"Breadth first search"

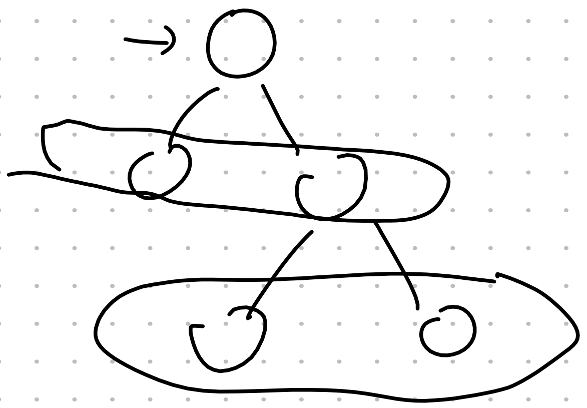
- exhaust all neighbors of a grid before moving onto neighbor of neighbors

DFS

- kept going until failure



```
queue<coord> q;
while q not empty {
    front = q.pop();
    push all of front's neighbors
}
```



[(1,2), (2,1), (1,3), (2,3), (1,4)]

still
a neighbor of 0