80 Pages
27.6 cm x 21.2 cm

Ruled 7 mm • Ligné 7 mm

EXERCISE BOOK
CAHIER D'EXERCICES

NAME/NOM _____

SUBJECT/SUJET CPSC 259

ASSEMBLED IN CANADA WITH IMPORTED MATERIALS
ASSEMBLÉ AU CANADA AVEC DES MATIÈRES IMPORTÉES

12107

Arrays

```
int age[100];
```

```
Const int Days = 365;
double temperatures[Days];
```

```
int Days = 365;
double temperatures[Days];
```
NO!

0	1	2	3	4	5	6	7	8	9	10

- Arrays can be intialized (filled) by using a loop

```
int marks[10];
for (int i = 0; i<10; i++)
    marks[i] = -1;
```

- or directly

```
int fib[] = {0, 1, 2, 3, 5, 8, 13};
```

~~arr1 = {1, 2, 3, 10};~~

NOT Allowed!

Ex. 2

Sum	index
0	1
4	2
12	3
28	4

Pointers

```
int a = 5;
int *p = &a;
```

- a is an integer variable with value 5
- p is a pointer variable storing the adress of a

Generic Pointer

```
Void* gp;
```

Opening / Reading / writing Files

- fgets function

```
Char* fgets(char* str, int n, FILE* stream)
```

- reads a line from the file and stores it in string
- stops at (n-1), \n, or end of file

- SScanf function

```
int sscanf(const char *str, const char *format,...)
```

Dynamic Memory

malloc function

ptr = (cast-type*) malloc (byte-size);

-returns a pointer to a memory block of at least size bytes

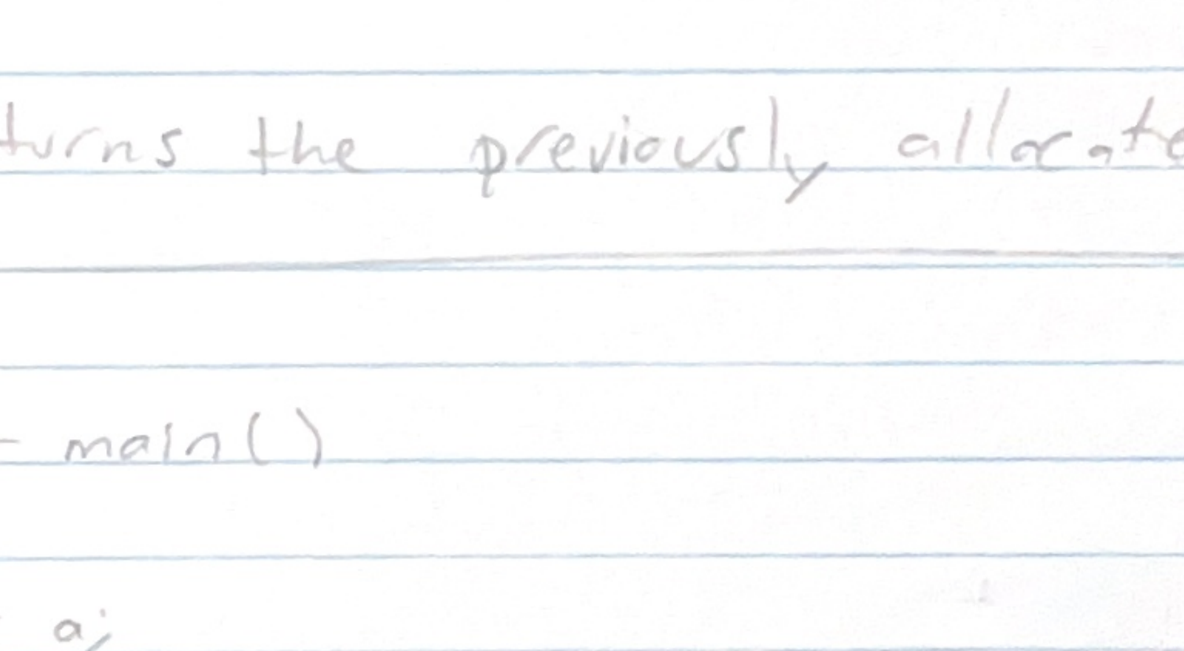
free function

free (ptr);

-returns the previously allocated memory block

18

```
int main()
{
int a;
int* p = (int*) malloc(sizeof(int));
*p = 10;
}
```



Efficiency

9/24/18
LEC

Code Runtime

- loop value n
- hardware
- compiler
- programming language

Running time

- function of n
- T(n)
- Doesn't depend on hardware, mathematical analysis

Order Notation

let T(n) and f(n) be functions mapping $Z^+ \to R^+$ (Positive ints)

T(n) $\in$ O(f(n)) if there are constants (and no such that $T(n) \leq C \cdot f(n)$ for all $n \geq n_0$

Meaning

T(n) is in big O of f(n)

T(n) will at least be better than f(n) which is a function we understand

$T(n) \in \Omega(f(n))$
$T(n) \geq c \cdot f(n)$

"Growth no better than f(n)"

$T(n) \in O(f(n))$
$T(n) \leq c f(n)$

"No worse than f(n)

$T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \Omega(f(n))$

"T(n) grows at the same rate as f(n)

Eliminate low order terms

Linked Lists

10/04/18

Unordered Arrays

inserting an item → O(1)

Taking an element out → options:

① shift everything down 1
Pro: order preserved
Con: costly if lots after

② Take the last one and put it there
pro: cheap con: order not preserved

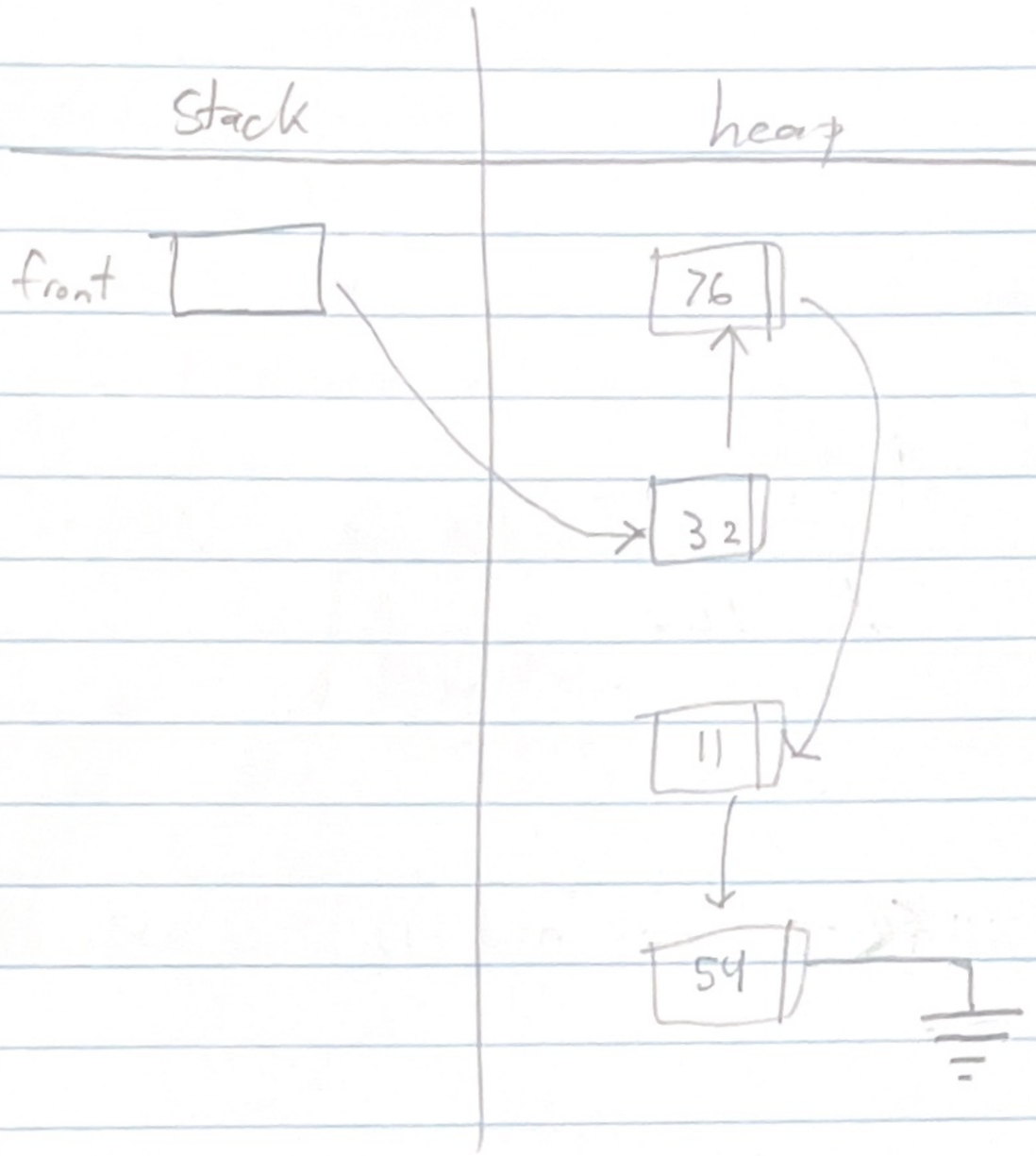
Ordered Arrays

inserting an item → O(n)

Linked List

- made up of linked nodes
- In the heap

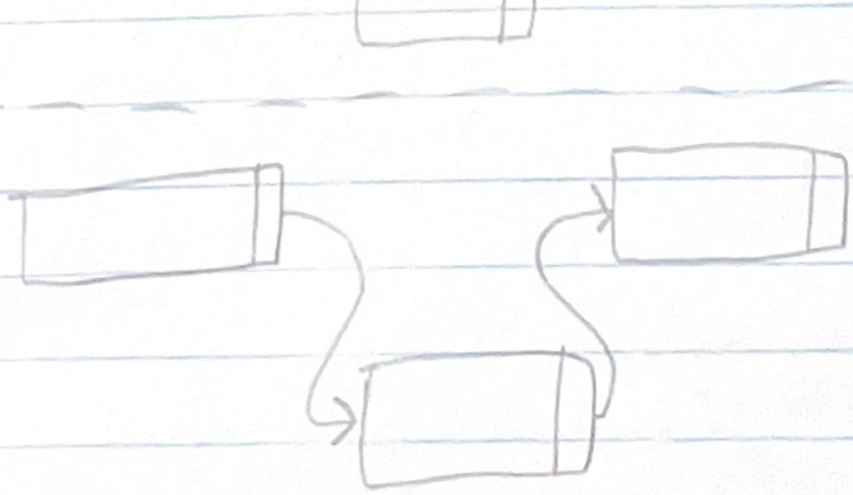
```
typedef struct node {
    int data;
    struct node* next;
} node;
```

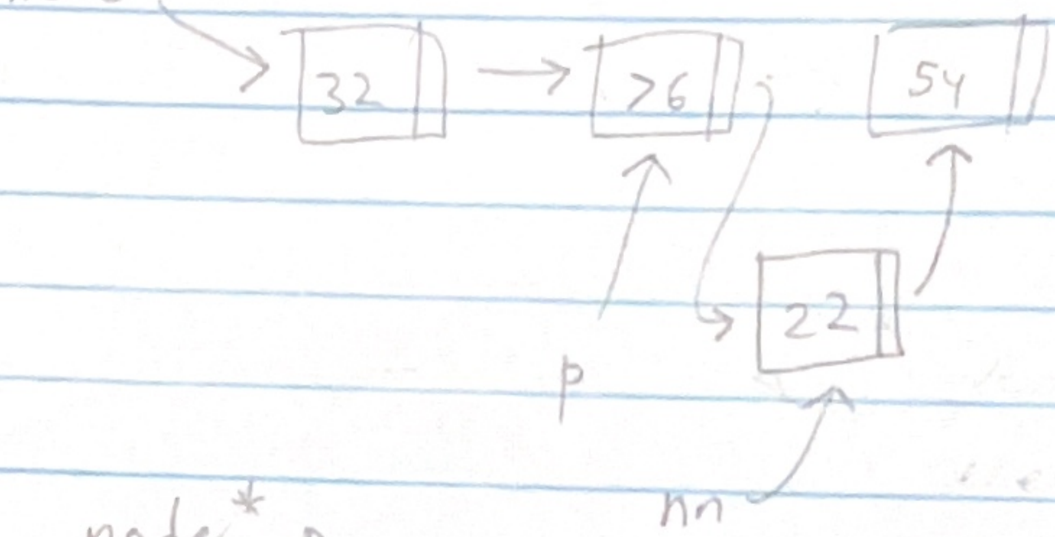


How do you know you're at the end?

last node has a pointer set to NULL

Inserting into a linked List



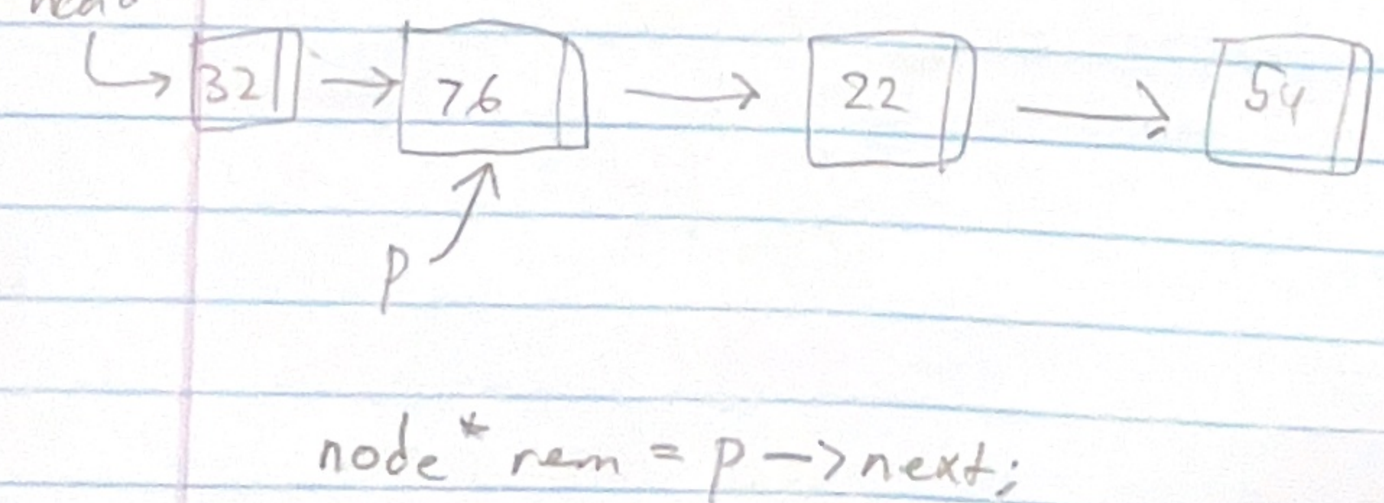



```
typedef struct node {
    int data;
    struct node *next;
} node;
```

* want to insert item

```
node* p;
p = head -> next;

node *nn = (node*) malloc(sizeof(node));

nn -> data = 22;
// *nn.data = 22; is equivilant

nn -> next = p -> next;
p -> next = nn;
```

Removing an item



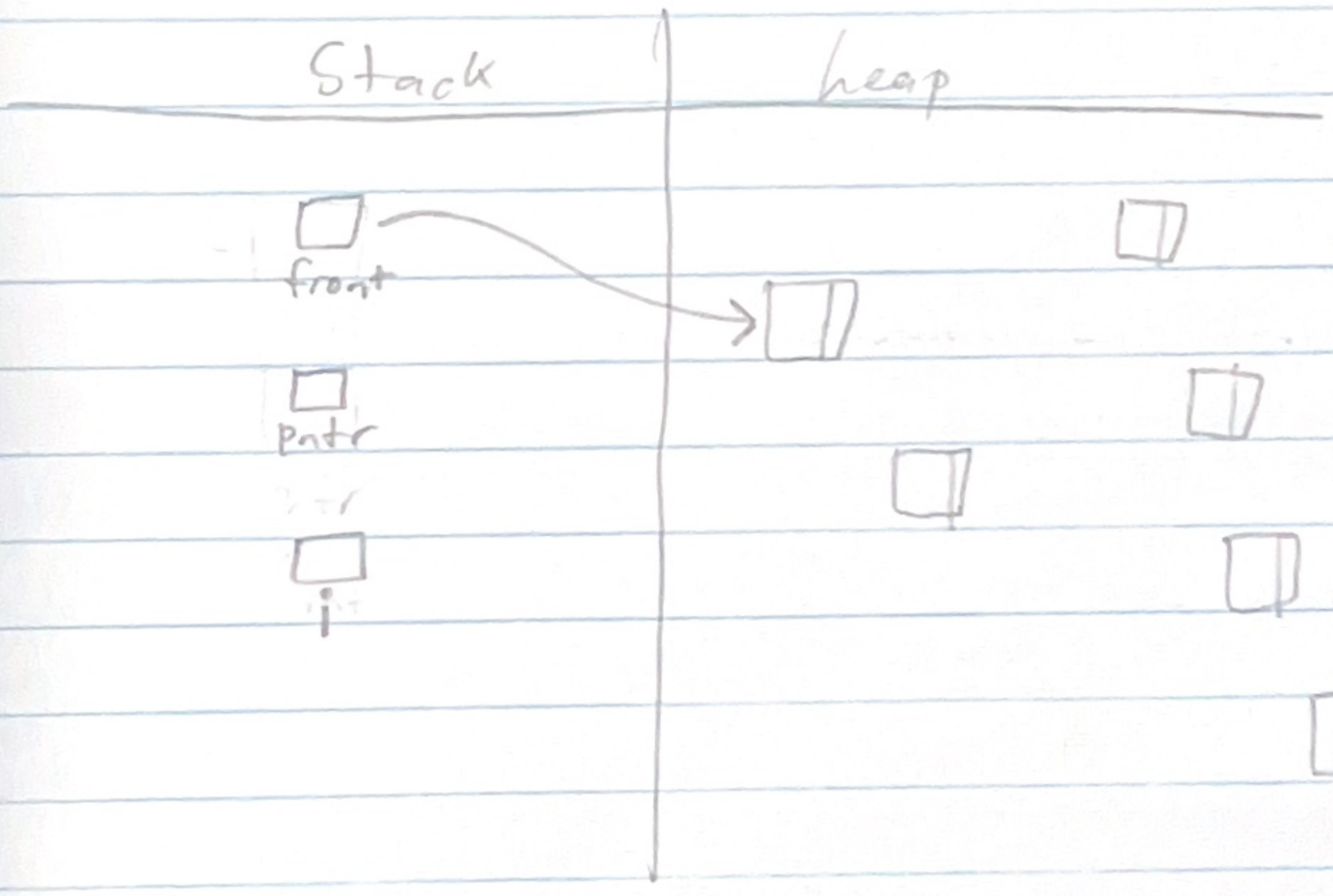

```
node* rem = p -> next;

p -> next = p -> next -> next;
        // (*(*p.next)).next   is equivilant

free(rem);
```

```
node* p = head;

while( .... ){
p = p->next;
}
```

To traverse the linked list to get to a specific node

Linked List

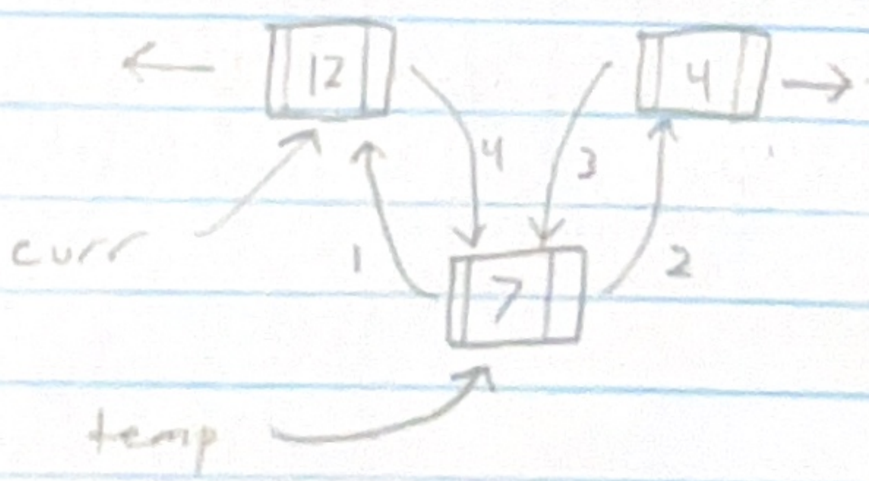
Linked List traversal




```
node* ptr = front;
int i;
for(i=0; i<4; i++){
  ptr = ptr->next;
}
```

Doubly-linked List

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



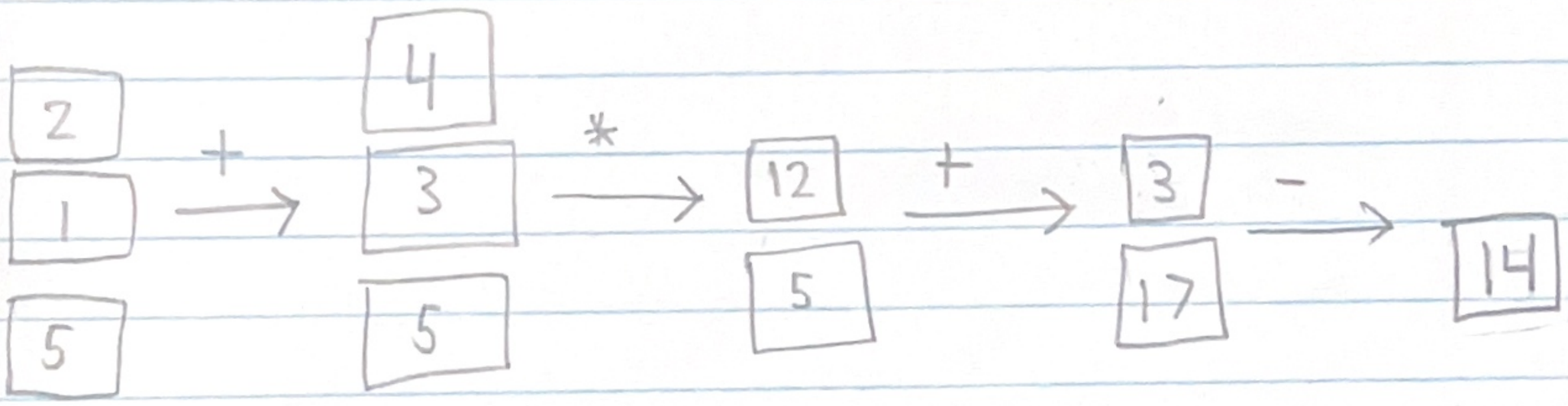
Abstract Data Types

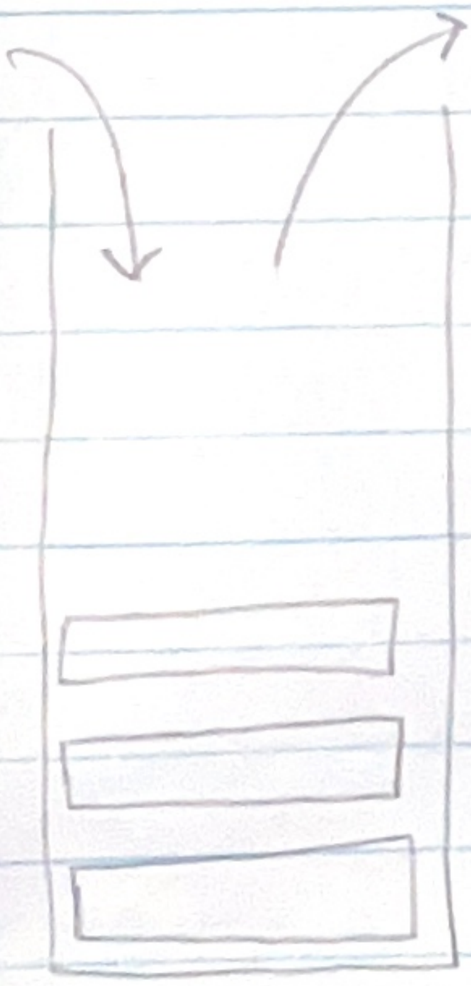
13+7 → Infix

13 7 + → Postfix :/ Reverse Polish Notation (RPN)

RPN

5 1 2 + 4 * + 3 -





- First in last out
- stack

- Stack is an ADT

Stack operations

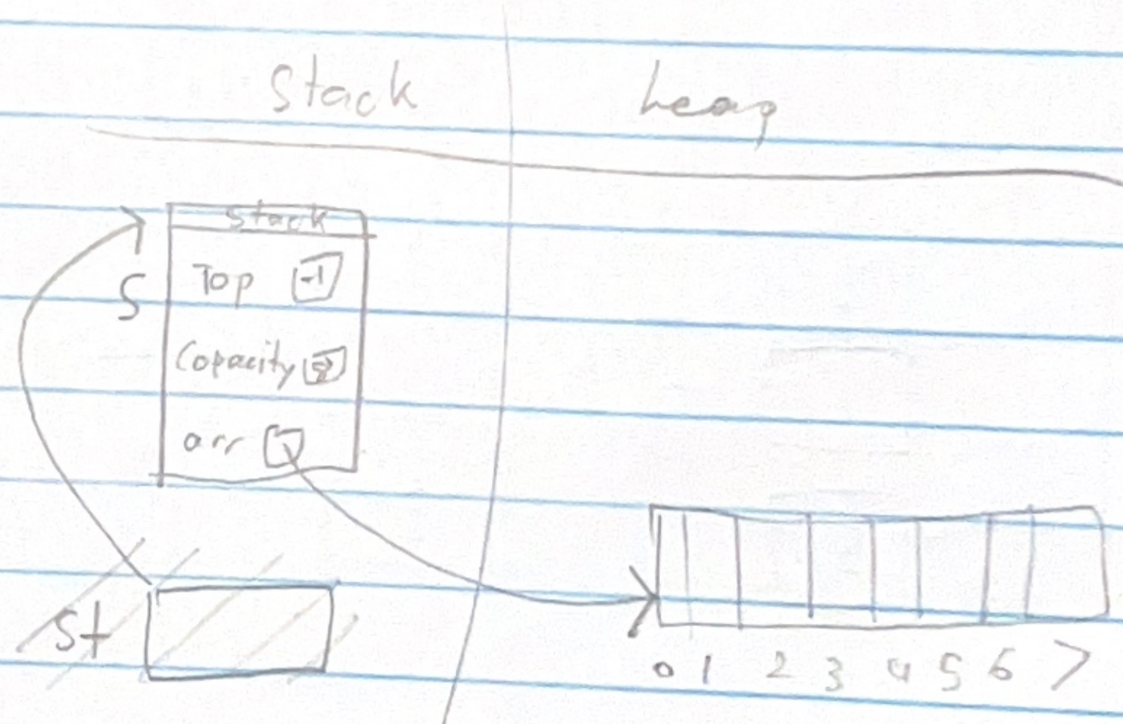
① Push - insert an item at the top of the stack

② Pop - remove and return the top item

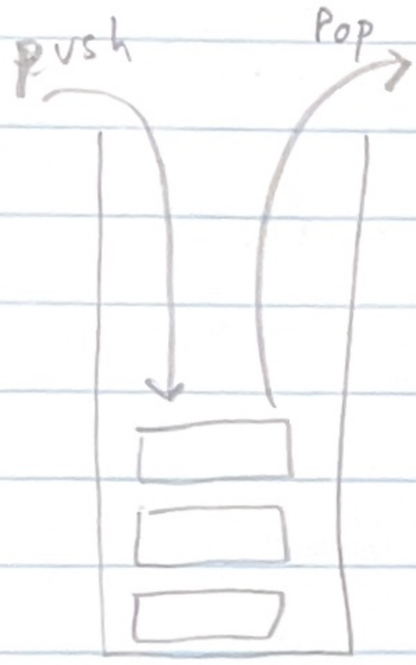
③ Peek - return the top item

④ Is Empty - does the stack contain any items?

```
main()
{
  Stack s;
  initialize(&s);
  push(&s, 3);
```

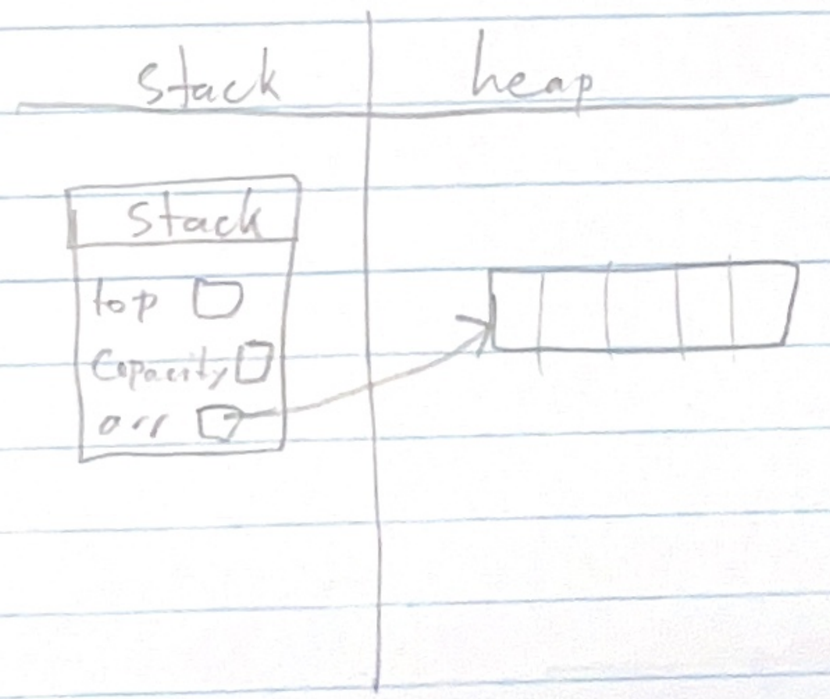


Abstract Data Types




```
typedef struct stack {
    int top;
    int capacity;
    int* arr;
} stack
```

```
stack s;
initialize_stack(&s);
push(&s, 7);
```



Complexity of push operation

(I think) O(n) CHECK!! O(1) per push
for 2n operations

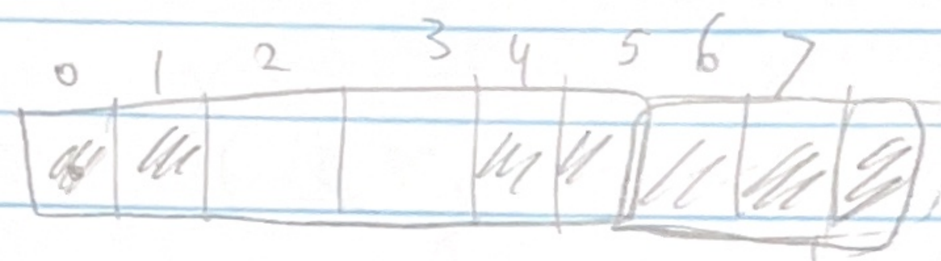
~~Build~~ Make room by *constant factor → O(1) BETTER
Make room by +constant → O(n) WORSE

Queue

10/11a/18

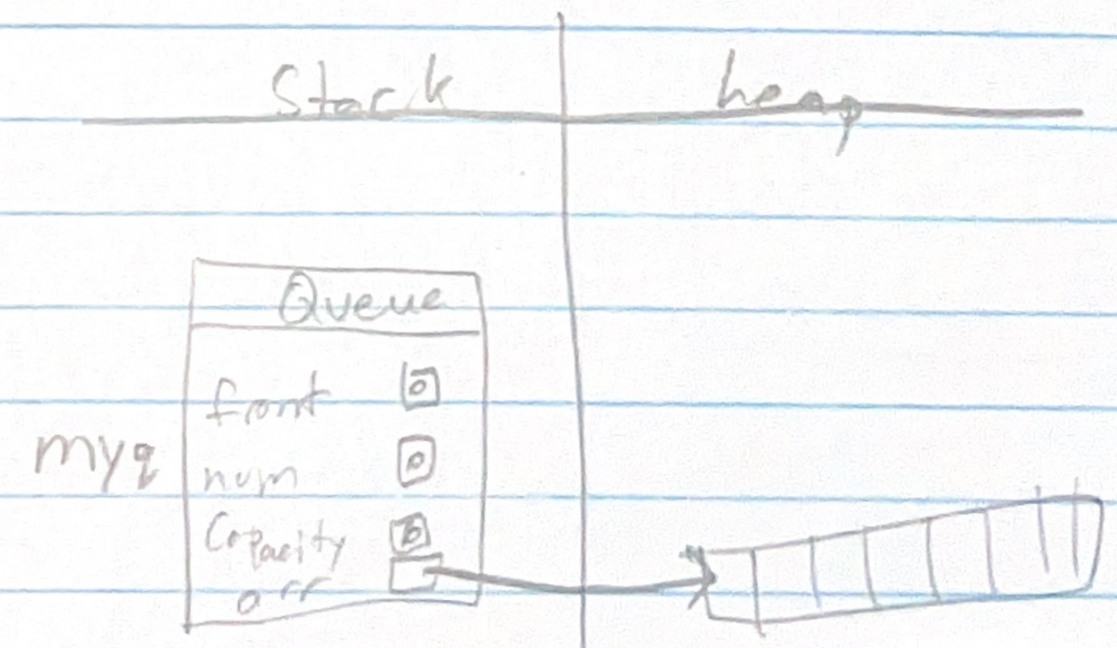
LEC

```
typedef struct{
  int front;
  int num;
  int capacity;
  int* arr;
} Queue;
```



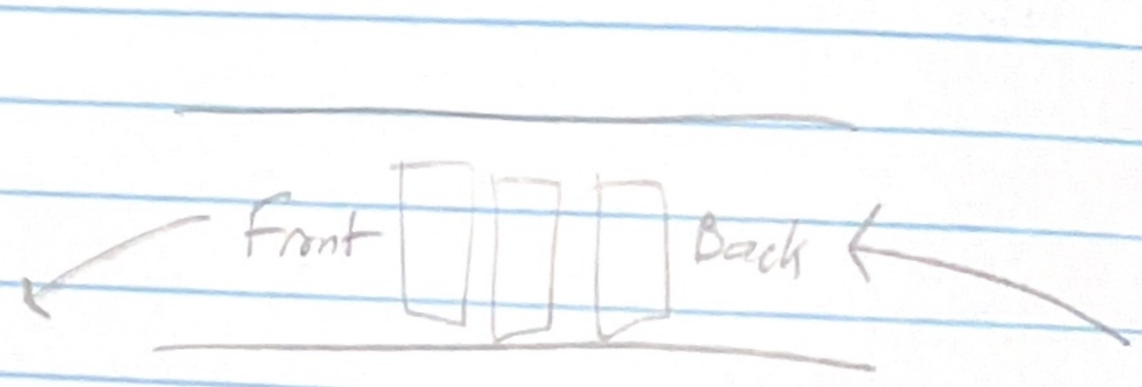
(front+num)%capacity = 2

```
Queue myq;
initQueue(&myq);
```



10/22/18
LEC

Abstract Data Types



Queue

Commands

Peek: returns data from front of the queue

Recursion

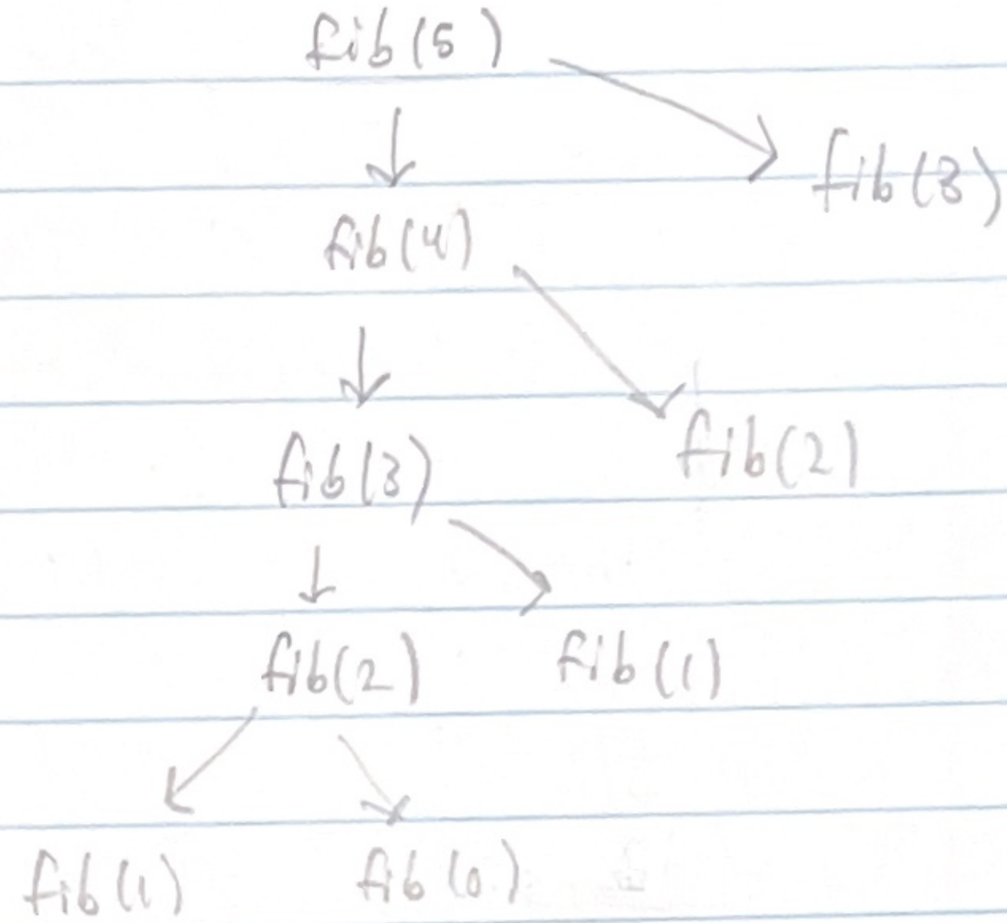
How many Rabbit pairs? Adults are immortal

Month	1	2	3	4	5	6
Baby	1	0	1	1	2	3
Adult	0	1	1	2	3	5
Total	1	1	2	3	5	8

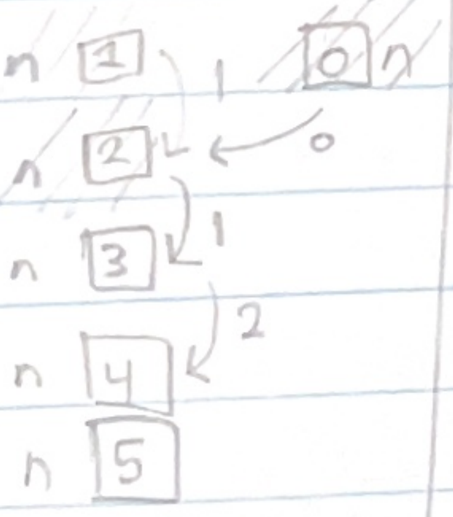
Fibonacci

- function can call itself
- must give it parameters that are smaller each time (Fibonacci)

```
int fib(int n)
{
  if (n<=0)
     return 0;
  else if (n==1)
     return 1;
  else
     return fib(n-1) + fib(n-2);
}
```



Stack	heap



Recursion

10/24/18
LEC

Find max in array / Iterative method

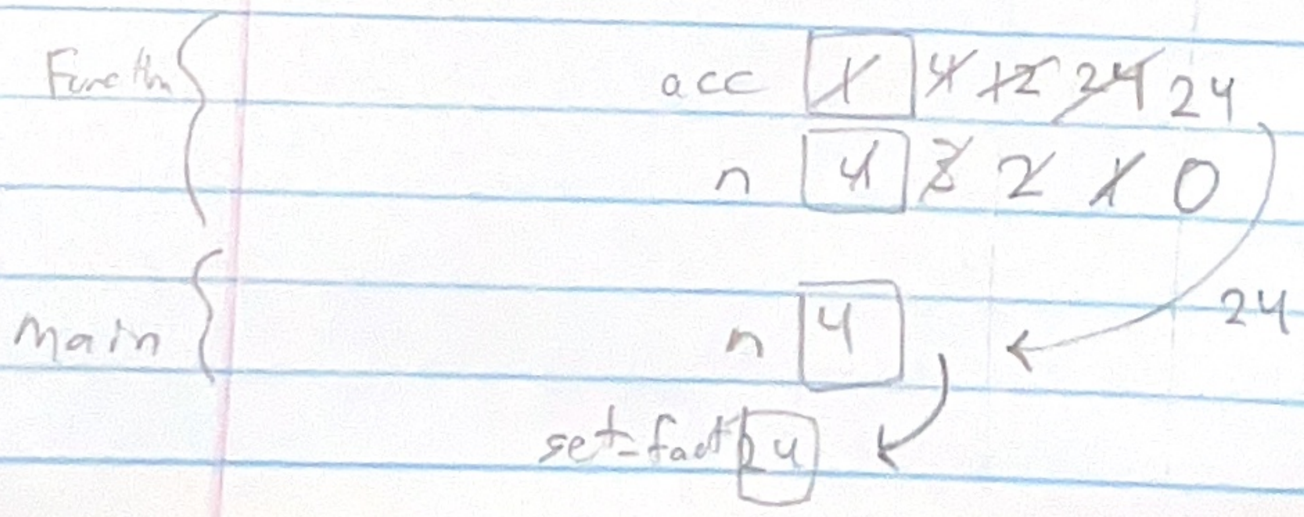
```
int findMaxIter(int arr[], int n)
{
  int maxval=arr[0];
  int i;

  for(i=1;i<n;i++)
    {
      if (arr[i]>maxval)
        maxval=arr[i];
    }
  return maxval;
}
```

Recursion functions:

1. Factorial
2. Find max
3. summation

Tail Factorial Recursive

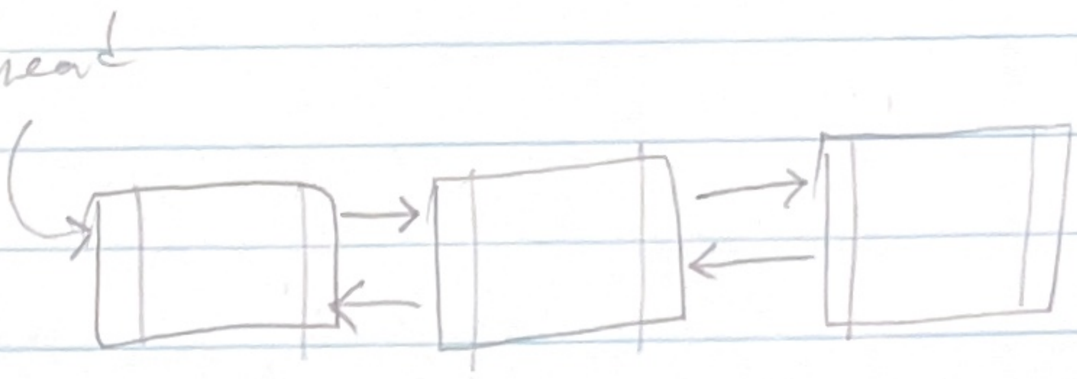
Stack



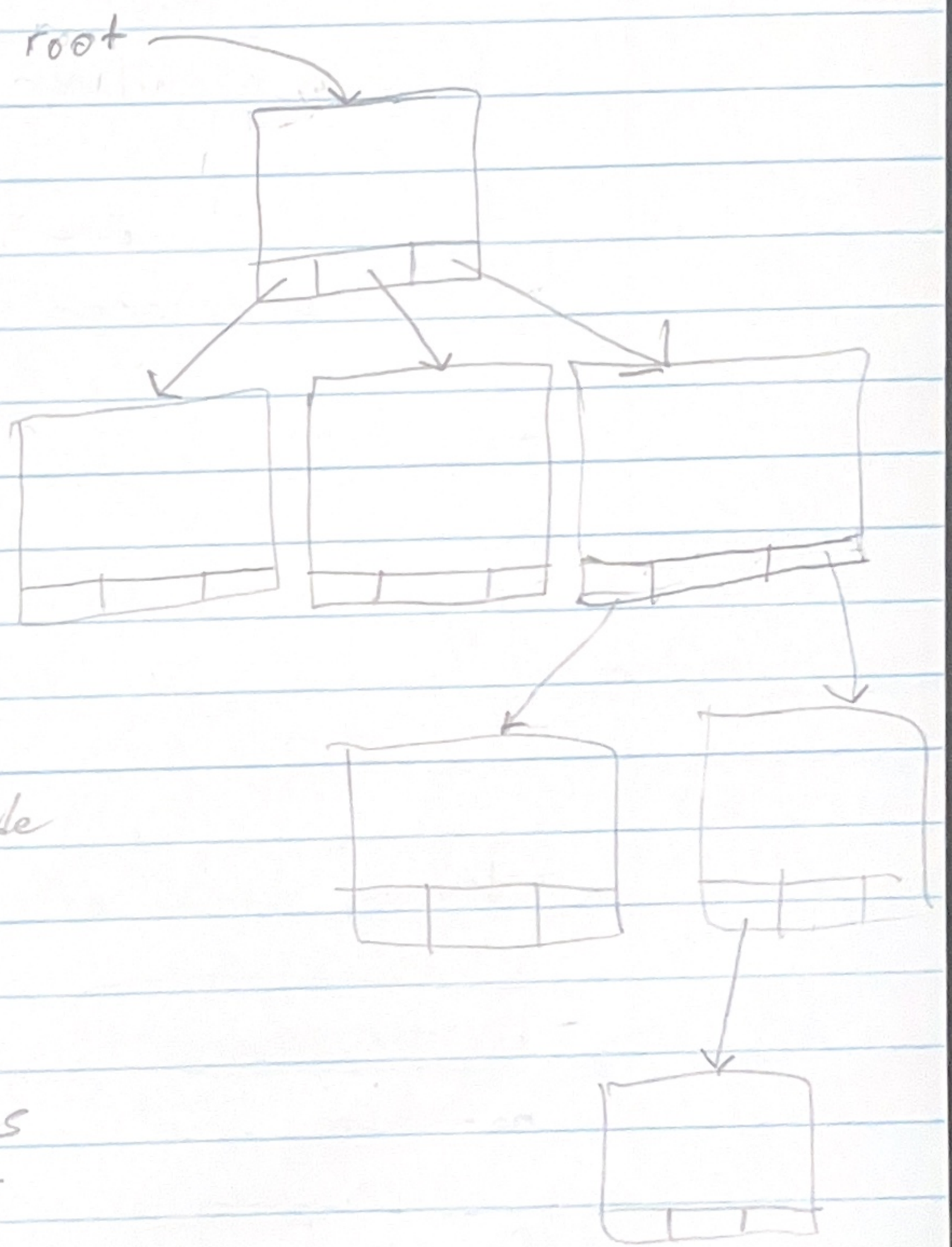
Abstract Data type

Trees

```
typedef struct LLnode{
    int data;
    LLnode* prev;
    LLnode* next;
} LLnode;
```

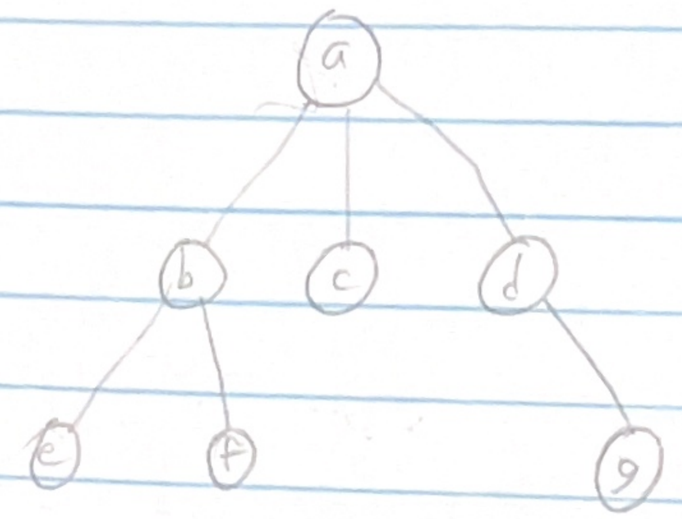



```
typedef struct treenode{
    int data;
    treenode* c1;
    treenode* c2;
    treenode* c3;
} treenode;
```



- To be a "tree", you must be able to get to every node through the root

- To be a "tree", it can have ~~most~~ number of edges as one less than number of nodes



- a is a parent of bcd
- bcd are children of a

- e and f are descendents of a
- b, c and d are siblings

- a and d are ancestors of g

- **Leaf**, node with no children

- Can have trees within trees

- **degree** number of children a node has

- **Branching factor**, the max # of children a node can have

- **Binary tree**, nodes with 2 children

- height = 1 + max(height(left), height(right));

recursive function

Binary Search Trees

18

Insertion

① create new node in dynamic memory

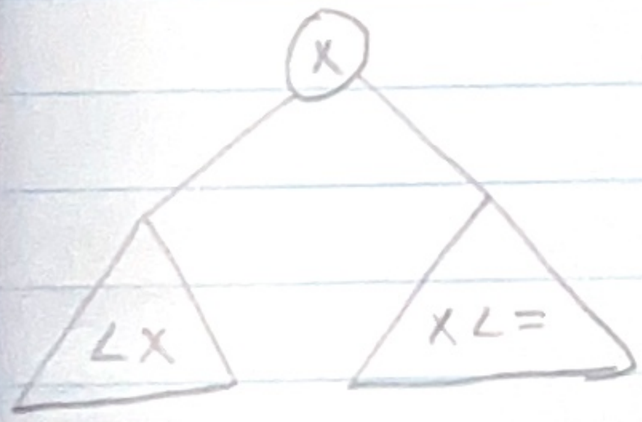
18

Priority Queue

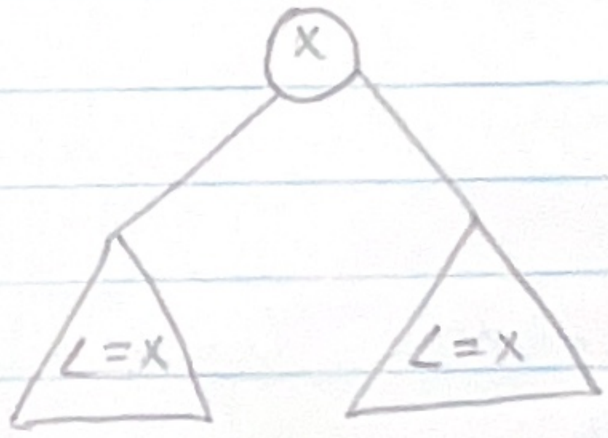
Binary heap

- Complete tree
- partially ordered

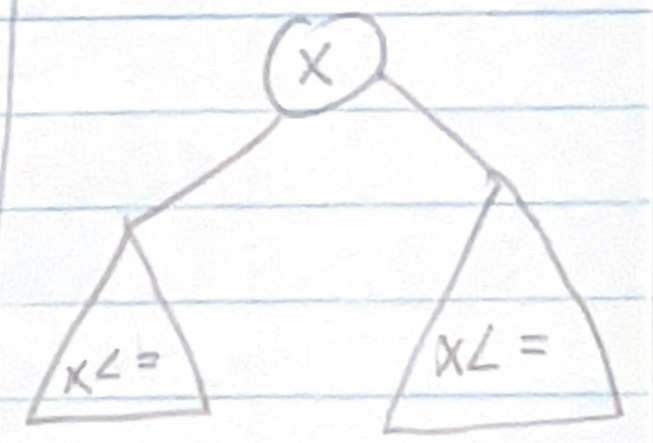
Full order



partial order (min)



partial order (max)



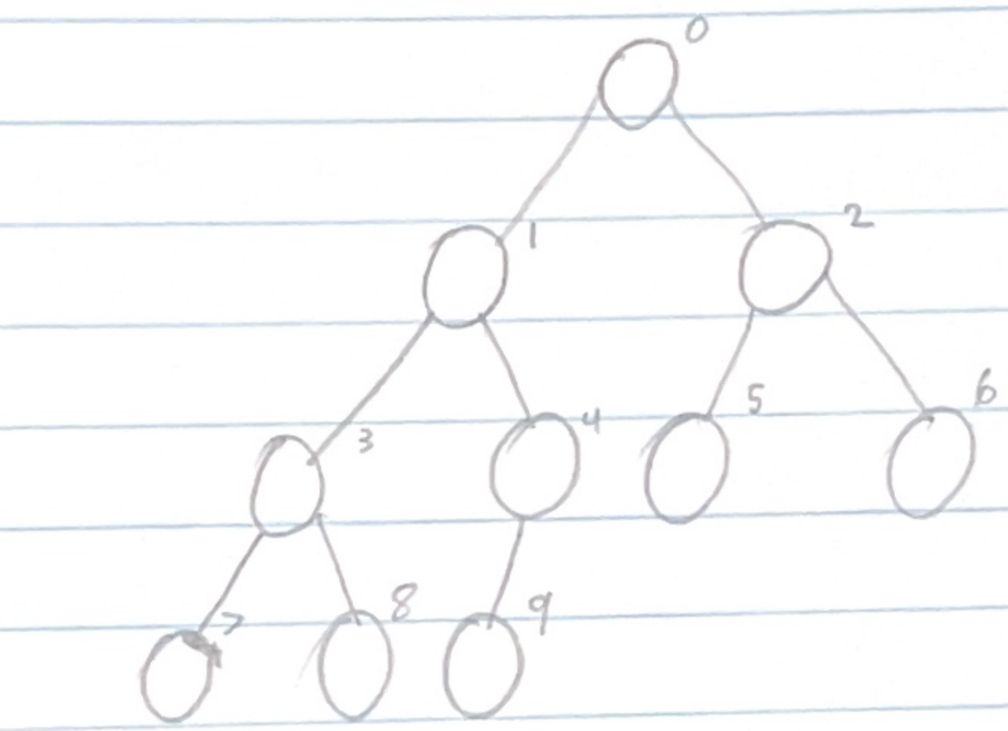
- Implented using arrays

Heap Insertion

Time Complexity

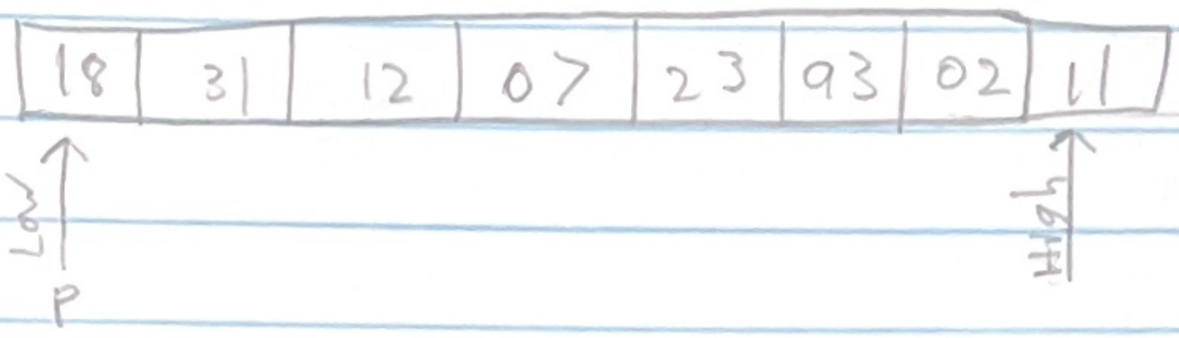
unordered array → min heap → ordered array

O(nlogn)



Quicksort

- Works by partioning the array



~~Pointers (Low (High (Pivot)~~

- Low
- High
- P

} indexes (int)

Best case	Worst case
O(nlogn)	n^2

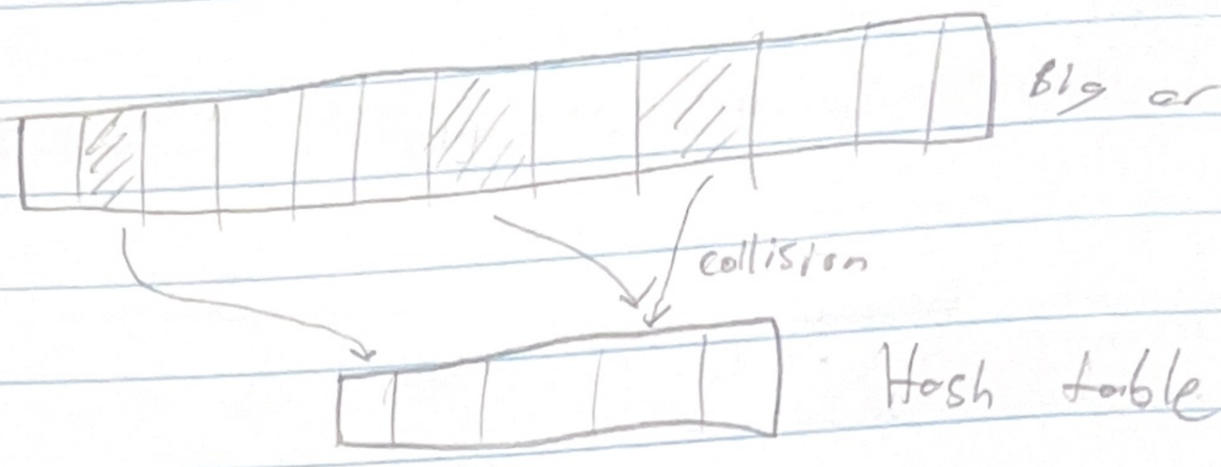
Average Case
O(nlogn)

Loop

```
while ( low != high )
  {
    while (arr[high] > arr[p])
        //scan high to the left.

  //Swap, set   p   to high index

    while (arr[low] < arr[p])
      //scan low to the right

    //swap, set p to low index

  }
```

Hash Tables

11/26/18
LEC



Collision

$f(x_1) = f(x_2)$ → collision will occur

Hash function properties

① Fast → want O(1) complexity
② Deterministic → not random
③ Uniformity

- Convert "strings" → int value

hash function ⟶ h(x) = x % tablesize

Deal with collisions how?

① Open addressing
② Separate chaining

Open addressing

- You go to put your (struct, value) in the hash table but there is already something there. So..... sol'n probe array (linear) find another free space