Stack: Linked List

Stack initStack() {

Stack s = (Stack) malloc(sizeof(StackType));

s->top = NULL; return s; }

int empty(Stack S) {

return (S->top == NULL); }

void push(Stack S, StackData d) {

Node \* n = (Node \*)malloc(sizeof(Node));

n->data = d; n->next = S->top; S->top = n; }

StackData pop(Stack S) {

if(empty(S)) exit(1); *// check empty*

StackData toReturn = S->top->data;

Node \* tmp = S->top;

S->top = S->top->next;

free(tmp);

return toReturn; }

StackData peek(Stack S) {

if(empty(S)) return BAD;

return S->top->data; }

void freeStack(Stack S) {

while(!empty(S)) pop(S);

free(S); }

Queue: Array – head points 1 before head value

Queue initQueue() {

Queue q = malloc(sizeof(QueueType));

q->head = 0; q->tail = 0; return q; }

int empty(Queue Q) { return (Q->head == Q->tail); }

void enqueue(Queue Q, QueueData d) {

if (full(Q)) {

printf("Queue full can’t add\n"); return;

}

Q->tail++;

Q->tail = Q->tail % MAX\_Q; *// wrap*

Q->data[Q->tail] = d;

}

QueueData dequeue(Queue Q) {

if (empty(Q)) {

printf("queue empty\n"); exit(1);

}

Q->head++;

Q->head = Q->head % MAX\_Q; *// wrap*

return Q->data[Q->head]; }

int full(Queue Q) {return (Q->tail % MAX\_Q==Q->head-1);}

void freeQueue(Queue Q) { free(Q); }

int length(Queue Q) {

if (Q->head <= Q->tail)

return Q->tail - Q->head;

else

return MAX\_Q - (Q->head - Q->tail); }

Print bases

void printNumInBase(int base, int n) {

char digits[] = "0123456789ABCDEFGHIJKLMOPQRSTUVWXYZ";

if (n < 0) { printf("-"); printNumInBase(base, -1 \* n);

} else if (n < base) { printf("%c", digits[n]);

} else {

printNumInBase(base, n /base);

printNumInBase(base, n %base);

}}

Sieve of Eratosthenes: O(n log( log n))

int i, j; // make sure numbers are positive!!!

int max = atoi(argv[1]); int len = max + 1;

int prime\_count = 0;

bool \*hits = malloc(len \* sizeof(bool));

for (i = 0; i < len; i++)

hits[i] = false; *// clear hits array*

*// it. In arr till sqrt(len)*, *mark off multiples of primes*

for (i = 2; i \* i <= len; i++) {

if (!hits[i]) { *// mark off its multiples in hits*

for (j = i; j < len; j++) { hits[j \* i] = true; }

for (i = 2; i < len; i++) if (!hits[i]) printf("%d\n", i);

free(hits); // /\ prints primes

Russian Peasant Algorithm

int peasant(int a, int b) {

if (b == 1) return a;

if(b % 2 == 0) return peasant(a\*2, b/2);

return peasant(a\*2, b/2) + a; }

GCD

int gcd(int a, int b) {

if (b == 0) return a;

return gcd(b, a % b); }

Sum Digits

int sumDigits(int n) {

if (n == 0) return 0;

return sumDigits(n - 1) + n; }

**Palindrome ----------------**

int is\_palindrome(char \*str, int s, int e) {

if (s >= e) { return 1;

} else if (str[s] != str[e]) { return 0;

} else {

return is\_palindrome(str, s + 1, e - 1);

return result;

}

}

Merge sorted linked lists

Node \*merge(Node \*list1, Node \*list2) {

if (list1 == NULL) { return copyList(list2);

} else if (list2 == NULL) { return copyList(list1);

} else if (list1->num < list2->num) {

return makeNode(list1->num, merge(list1->next, list2));

} else {

return makeNode(list2->num, merge(list1, list2->next));

}}

Postfix calculator

for (i = 0; exp[i]; ++i) {

        if (isdigit(exp[i])) push(stack, exp[i] - '0');

        Else {

            int val1 = pop(stack);

            int val2 = pop(stack);

            switch (exp[i]) {

            case '+': push(stack, val2 + val1); break;

            case '-': push(stack, val2 - val1); break;

            case '\*': push(stack, val2 \* val1); break;

            case '/': push(stack, val2/val1); break;

            }

        }

    }

    return pop(stack); }

Linked List insert in order

void insertInOrder(int n, Node \*\* list\_ptr) {

Node \* list = \*list\_ptr;

while (list != NULL) {

if (n < list->num) { list = list->next;}

else { list->next = makeNode(n, list->next); return; }

} }

Reverse

void RecursiveReverse(struct node\*\* headRef) {

struct node\* first; struct node\* rest;

if (\*headRef == NULL) return; // empty list base case

first = \*headRef; rest = first->next;

if (rest == NULL) return;

RecursiveReverse(&rest);

first->next->next = first;

first->next = NULL;

\*headRef = rest; }

Pair Matching

int i;

Stack s = initStack();

for (i = 0; i < numread; i++) {

if ('[' == input[i] || '(' == input[i]) {

push(s, input[i]);

} else if (']' == input[i]) {

if (pop(s) != '[') { printf("no\n"); return; }

} else if (')' == input[i]) {

if (pop(s) != '(') { printf("no\n"); return; }

}

}

if (empty(s)) { printf("yes\n");

} else { printf("no\n"); }

Count occurrences in Linked List ------------------------

int countR(Node \* top, int n) {

if (top == NULL) return 0;

if (top->num == n) return count(top->next, n) + 1;

else return count(top->next, n);}

makeNode

Node \*makeNode(int n, Node \*nextItem) {

Node \*ret = (Node \*)malloc(sizeof(Node));

ret->num = n; ret->next = nextItem;

return ret; }

Insert head:

\*listPtr = makeNode(n, \*listPtr); // top points at new node

Insert Tail

void insertTail(int n, Node \*\*listPtr) {

Node \*list = \*listPtr;

if (list == NULL) {\*listPtr = makeNode(n, NULL); return; }

while (list != NULL) {

if (list->next == NULL)

list->next = makeNode(n, NULL);

return;

}

list = list->next;}}

Delete

int delete (Node \*toDelete, Node \*\*listPtr) {

Node \*list = \*listPtr;

if (toDelete == NULL || list == NULL) {return 0;}

if (toDelete == list)

\*listPtr = list->next; free(toDelete); return 1;

Node \*before = list; list = list->next;

while (list != NULL) {

if (toDelete == list)

before->next = list->next; free(list); return 1;

before = list; list = list->next;

}

return 0; *// toDelete not found* }

False is 0; True is != 0

O(log n) - binary search

for(i=0; i\*i < n; i++) { //something in constant time...}

O(n log n)

for (i = 0; i < n; i++) { // linear loop O(n) \* ...

for (j = 1; j\*j < n; j++) { // ...log (n)

// do something in constant time... }}

[array](https://github.com/alexweininger/notes/blob/master/C/assets/binsearch.PNG)

bin(int firstIndex, int lastIndex); // for 41

bin(0, 15); bin(8, 15);

bin(8, 11); bin(10, 11);bin(11, 11);

O(log n) – for (i=0; i<n; i\*=2)

binary search

O(n^(0.5)) - for(i = 0; i \* i < n; i++)

O(n)

single iteration

O(n log n)

heapsort

merge sort

Quick sort

O(n^2)

insertion sort

selectin sort

O(2^n)

recursion: towers of Hanoi, scheduling

O(n!)

O(n^3):

for(i = 0; i < n; i++)

for(j = 0; j < n; j++)

for(k = 0; k < j; k++)

O(n log n):

for(i = 0; i < n; i++)

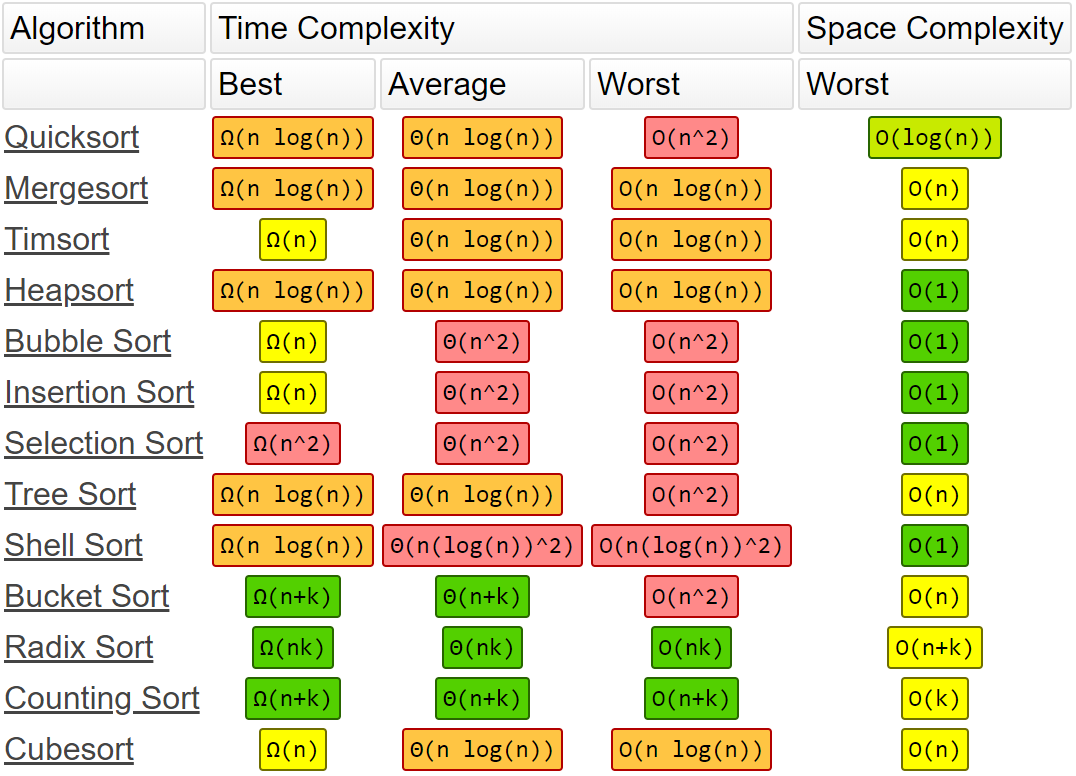
for(j = 1; j < n; j = j\*2)

Circular Linked List

Node \* top = NULL;

Node \* tail = makeNode(3, top);

makeNode(2, top);

makeNode(1, tail);

void printList(struct Node \*first) {

 struct Node \*temp = first;

 if (first != NULL)

    do { temp = temp->next; } while (temp != first); }

gdb:

info break, backtrace, cont, print, step, break [line], info frame

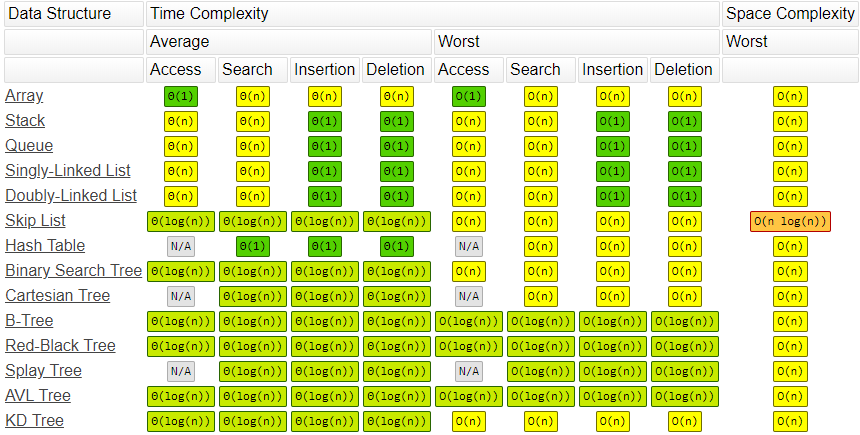
Hanoi:

void t(int n, char from\_rod, char to\_rod, char aux\_rod) {

    if (n == 1) return;

    t(n-1, from\_rod, aux\_rod, to\_rod);

    t(n-1, aux\_rod, to\_rod, from\_rod); }

QUEUES arrays:

ARR[SIZE] means the queue can hold 9 values but the data array which the queue values are stored in can be full (with 10 values). When full, the value currently pointed to by head is not actually in the queue. Queue is empty when head == tail, full when tail % SIZE == head – 1.

Repeated Div:

for (i = 2; i <= max; i++) {

int found\_prime = 1;

for (j = 2; j \* j <= i; j++) {

if (i % j == 0){ found\_prime = 0; break; }

}

if (found\_prime) printf("%d\n", i); // print prime

}