VIEWING FILES
Cd.. goes to parent dir
cd - goes to prev dir
Cd ~ goes to home dir
Pwd prints cd
Ls -a lists all files
(even with .)
Ls -F appends / for dir
Ls -t by last modified
Ls -s sorts by size
Ls - l long format
MOVING FILES
Mkdir creates dir
Mv moves
Cp copies
Cp f1 f1 copies f1 to f2
Cp f1 f2 d1 copies files to d1
Cp -r d1 d2 copies d1 to d2
Rm -r d1 removes directory
REDIRECTING STDIN
>file rewrites output to that file
>>file appedns output
Cat > file reads from stdin,
writes to file
PIPELINE
(e.g sort < bro.txt | uniq)
Sort, uniq, grep <- prints line
matching pattern
PERMISSIONS
Chmod
<three digit num 1-7> <filename>
FIND
Find <starting directory> <options> <expression>
Options: -type d, -type f
Expression: -name <string> -size <+5M>
Operators: -a for and, -o for or, ! for not
E.g find . -type f -name /(“.txt” -o “.pdf” /) -size +5M
BITWISE OPERATOR
& bitwise and
| bitwise or
^ bitwise xor
They compare each bit of
both binary versions of each num
a<<n means a*2^n in base10
a>>n means a/2^n in base10
ASCII
48-57 is ‘0’ to ‘9’
65-90 is ‘A’ to ‘Z’
97-122 is ‘a’ to ‘z’

 # COMPRESSION
tar -cvf f.tar dir # Archive
gzip file # Compress
gunzip file.gz # Decompress
zip f.zip file # Zip
unzip f.zip # Unzip
 # MISC
echo "text" # Print text
date # Show date/time
man cmd # Manual
alias ll="ls -la" # Shortcut
history # Command history
clear # Clear screen
 # GITHUB
cd path/my_project
git init
git add readme.txt or git add . (all)
git commit -m "Initial project version"
 # GDB - Breakpoints
break main # Break at start of main()
break 10 # Break at line 10
break file.c:25 # Break at line 25 in file.c
delete # Remove all breakpoints
delete 1 # Remove breakpoint ID 1
enable 1 / disable 1 # Turn breakpoint on/off
info breakpoints # List all breakpoints
 # GDB - Execution
run # Start the program
continue / c # Resume after hitting a breakpoint
next / n # Step over (skip into function)
step / s # Step into function
finish / fin # Run until current function returns
 # GDB - Variables
info locals # Show local variables
print x # Print value of x
print arr[2] # Print element at index 2
set var x=10 # Set x to 10
watch x # Break when x changes
 # GDB - Call Stack
backtrace / bt # Show call history (function stack)
frame 1 / f 1 # Go to frame 1
 # GDB - Code View
list / l # Show lines of code
layout src # Split view (code + gdb)
refresh # Redraw screen
 # Valgrind - Memory Leaks
gcc -g file.c # Compile with debug info
valgrind ./a.out # Basic memory check
valgrind --leak-check=full ./a.out # Full leak info
 # gprof - Performance Profiling
gcc -pg file.c # Compile for profiling
./a.out # Run program to generate gmon.out
gprof ./a.out gmon.out # Show profiling info

 # Integer Types
char 1 B %c [-128 to 127]
short 2 B %hd [-32K to 32K]
int 4 B %d/%i [-2.1B to 2.1B]
long 8 B %ld [-9.2Q to 9.2Q]
long long 8 B %lld same as long
 # Unsigned Integer Types
unsigned char 1 B %c [0 to 255]
unsigned short 2 B %hu [0 to 65K]
unsigned int 4 B %u [0 to 4.2B]
unsigned long 8 B %lu [0 to 18.4Q]
unsigned long long 8 B %llu same as above
 # Floating Point Types
float 4 B %f ~6-7 decimal digits
double 8 B %lf ~15 decimal digits
long double 16 B %Lf ~18+ digits
 # Pointers
void* 8 B %p memory address (hex)
int size = sizeof(array)/sizeof(array[0])
 # GCC BASICS
gcc -c file.c # Compile file.c to object file (file.o)
gcc file.o -o app #Link object file to make executable
gcc file.c -o app # Compile + link in one step
 # MULTIPLE FILES
gcc -c main.c util.c #Compile multiple files to .o files
gcc main.o util.o -o app # Link .o files into one exe
gcc main.c util.c -o app # Compile & link in one step
 # COMMON FLAGS
-Wall # Show all common warnings
-g # Include debug info for gdb/valgrind
-c # Compile only, no linking
-DNAME=val# Define a macro like #define NAME
-std=c11 # Use C11 standard
-Idir # Add include path
-Ldir # Add library path

 # POINTER BASICS
int x = 5;
int* p = &x; // p holds address of x
*p = 10; // change value at address (x becomes 10)
printf("%d", *p); // dereference to get value
 # DECLARATION
int* p; // pointer to int
 # POINTER & ARRAYS
int arr[3] = {1, 2, 3};
int* p = arr;
p[1] // same as arr[1]
*(p + 2) // same as arr[2]
 # POINTER TO POINTER
int x = 5;
int* p = &x;
int** pp = &p;
**pp // value of x
*pp // address of x
pp // address of p

 # DYNAMIC MEMORY (HEAP)
int* a = (int*)malloc(4 * sizeof(int));
int* b = (int*)calloc(4, sizeof(int));
a = (int*)realloc(a, 8 * sizeof(int));
# resize memory block (‘a’ hold 8 ints)
free(a);
 # NOTES
- check for NULL after allocation
- realloc can move memory; reassign

STRING METHODS <string.h>
strcat(s,t) //concatenate t to end of s
strncat(s,t,n) //concatenate n characters of t to end of s
strcmp(s,t) //return negative, zero, or positive for s < t, s == t, s > t
strncmp(s,t,n) //same as strcmp but only in first n characters
strcpy(s,t) //copy t to s
strncpy(s,t,n) //copy at most n characters of t to s
strlen(s) //return length of s
strchr(s,c) //return pointer to first c in s, or NULL if not present
strrchr(s,c) //return pointer to last c in s, or NULL if not present

DEEP COPY
```
struct BookNode* deepCopyBookList(struct BookNode* original) {
    if (!original) return NULL; // If the original list is empty, return NULL

    struct BookNode* newHead = NULL;  // Head of the copied list
    struct BookNode* newTail = NULL;  // Pointer to track the last node in the new list

    while (original) {
        // Allocate memory for a new BookNode
        struct BookNode* newNode = (struct BookNode*)malloc(sizeof(struct BookNode));

        // Deep copy: Allocate memory for the data and copy the content
        newNode->data = malloc(sizeof(*(original->data))); // Allocate memory for the data

        *(newNode->data) = *(original->data); // Copy the contents of data

        newNode->next = NULL; // Initialize next to NULL

        // If this is the first node, set it as the head
        if (!newHead) {
            newHead = newNode;
            newTail = newNode;
        } else {
            newTail->next = newNode; // Append to the new list
            newTail = newNode;       // Move tail forward
        }

        original = original->next; // Move to the next node in the original list
    }
    return newHead; // Return the head of the copied list
}
```

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

ADD BEFORE HEAD
```
void insertAtStart(Node **root, int
value)
{
    Node *new_node =
malloc(sizeof(Node));
    new_node->data = value;
    new_node->next = *root;
    *root = new_node;
}
```

APPEND TO TAIL
```
void insertAfter(Node *node, int value)
{
    Node *new_node =
malloc(sizeof(Node));
    new_node->data = value;
    new_node->next = node->next;
    node->next = new_node;
}
```

INSERT AFTER A NODE
```
void insertAfter(Node *node, int value)
{
    Node *new_node =
malloc(sizeof(Node));
    new_node->data = value;
    new_node->next = node->next;
    node->next = new_node;
}
```

INSERT SORTED
```
void insertSorted(Node **root, int value)
{ if (*root == NULL || (*root)->data >=
value){insertAtStart(root, value);
        return;}
Node *curr = *root;
while (curr->next != NULL){
   if (curr->next->data >= value){ i
insertAfter(curr, value);
            return;}
   curr = curr->next;}
    insertAfter(curr, value);}
```

FIND NODE BY VALUE
```
ListNode* findNode( ListNode* head, int key) {
    for ( ListNode* cur = head; cur; cur =
cur->next) {
        if (cur->data == key) return cur;
    } return NULL;}
```

REVERSE LIST
```
ListNode* reverse( ListNode* head) {
    if (head == NULL || head->next == NULL)
return head;
    ListNode *pre = NULL, *cur = head;
    while (cur) {
        ListNode* next = cur->next;
        cur->next = pre;
        pre = cur;
        cur = next;
    } return pre;}
```

REMOVE ELEMENT BY VALUE
```
void removeElement(Node **root, int value)
{
    if (*root == NULL)
    {
        return;
    }
    if ((*root)->data == value)
    {
        Node *to_remove = *root;
        (*root) = (*root)->next;
        free(to_remove);
        return;
    }

    for (Node *curr = *root; curr->next != NULL;
curr = curr->next)
    {
        if (curr->next->data == value)
        {
            Node *to_remove = curr->next;
            curr->next = curr->next->next;
            free(to_remove);
            return;
        }
    }
}
```

DEALLOCATE
```
void deallocate(Node **root)
{
    Node *curr = *root;
    while (curr->next != NULL)
    {
        Node *to_free = curr;
        free(to_free);
    }
    *root = NULL;
}
```

#STRUCT
```
typedef struct { int id; char name[50]; } Student;
typedef struct TreeNode {
    int data;
    struct TreeNode** children;
} TreeNode;
typedef struct BTreeNode {
    int data;
    struct BTreeNode* left;
    struct BTreeNode* right;
} BTreeNode;
typedef struct BSTNode {
    Student data;
    struct BSTNode* left;
    struct BSTNode* right;
} BSTNode;
```
#INSERT (RECURSIVE)
```
BSTNode* insert(BSTNode* root, BSTNode* node) {
    if (!root) return node;
    if (node->data.id <= root->data.id) {
        root->left = insert(root->left, node);
    } else {
        root->right = insert(root->right, node); }
    return root; }
```
#INSERT (ITERATIVE)
```
BSTNode* insert_iterative(BSTNode* root, BSTNode* node) {
    if (!root) return node;
    BSTNode* cur = root;
    while (cur) {
        if (node->data.id <= cur->data.id) {
            if (!cur->left) {
                cur->left = node;
                break;}
            cur = cur->left;
        } else {
            if (!cur->right) {
                cur->right = node;
                break;}
            cur = cur->right;}
    }return root;}
```
#PREORDER
```
void preorder(BSTNode* root) {
    if (!root) return;
    printf("Student %s id: %d\n", root->data.name, root->data.id);
    preorder(root->left);
    preorder(root->right);}
```
#POSTORDER
```
void postorder(BSTNode* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
    printf("Student %s id: %d\n", root->data.name, root->data.id);}
```
#CLONE
```
BSTNode* clone(BSTNode* root) {
    if (!root) return NULL;
    BSTNode* cloned = malloc(sizeof(BSTNode));
    cloned->data.id = root->data.id;
    strcpy(cloned->data.name, root->data.name);
    cloned->left = clone(root->left);
    cloned->right = clone(root->right);
    return cloned;}
```
#COMPARE
```
bool same(BSTNode* root1, BSTNode* root2) {
    if (!root1 || !root2) return root1 == root2;
    if (root1->data.id != root2->data.id ||
        strcmp(root1->data.name, root2->data.name) != 0) {
        return false;}
    return same(root1->left, root2->left) &&
           same(root1->right, root2->right);}
```
#HEIGHT
```
int height(BSTNode* root) {
    if (!root) return 0;
    return 1 + max(height(root->left), height(root->right));}
```
#SIZE
```
int size(BSTNode* root) {
    if (!root) return 0;
    return 1 + size(root->left) + size(root->right);}
```
#DELETE
```
BSTNode* delete(BSTNode* root, int id) {
    if (!root) return NULL;
    if (id < root->data.id) {
        root->left = delete(root->left, id);
    } else if (id > root->data.id) {
        root->right = delete(root->right, id);
    } else {
        if (!root->left) {
            BSTNode* r = root->right;
            free(root);
            return r;}
        if (!root->right) {
            BSTNode* l = root->left;
            free(root);
            return l;}
        BSTNode* suc = root->right;
        while (suc->left) {
            suc = suc->left;}
        root->data.id = suc->data.id;
        strcpy(root->data.name, suc->data.name);
        root->right = delete(root->right, suc->data.id);}
    return root;}
```
#FREE TREE
```
void freeBST(BSTNode* root) {
    if (!root) return;
    freeBST(root->left);
    freeBST(root->right);
    free(root);}
```