./README.md

```markdown
## Data Structures: C implementations

[**Dynamic Arrays**](./dynamic-array/dynamic-array.c): Dynamically Resizing Array with variable size<br/>
[**Stack**](./stack/stack.c): First-In-Last-Out Data Structure <br/>
[**Queue**](./queue/queue.c): First-In-First-Out Data Structure<br/>
[**Trees**](./tree): Hierarchy based data structure <br/>
[**Generic Tree**](./tree/tree-generic/tree_generic.c): Tree with variable children for each node <br/>
[**Binary Tree**](./tree/tree-binary/tree_binary.c): Tree with maximum two children<br/>
[**Complete Binary Tree**](./tree/tree-binary-complete/tree_binary_complete.c): Tree with stricty two children, filled in pre-order<br/>
[**Heap**](./tree/heap/heap.c): Complete Binary Tree based priority queue<br/>
[**Graph**](./graph/graph.c): data structure with vertices and edges<br/>
```

./tree/tree-binary/tree_binary.c

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node node;
struct node {
  int data;
  node* left;
  node* right;
};

typedef struct tree_binary tree;
struct tree_binary{
  node* root;
};

int main(){
  tree _t;
  tree* t = &_t;
  t -> root = NULL;
  return 0;
}
```

./tree/tree-binary-complete/tree_binary_complete.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "includes/tree-binary-complete.h"

void tree_initialize(tree* t){
  t -> size = 0;
  t -> capacity = 0;
  t -> contents = NULL;
}

void tree_reallocate(tree* t){
  if(t -> capacity == 0){
    t -> contents = (int *) malloc(sizeof(int));
    t -> capacity = 1;
  } else {
    t -> contents = (int *) realloc(t -> contents, sizeof(int) * ( t -> capacity * 2 ));
    t -> capacity *= 2;
  }
}

void tree_add_node(tree* t, int data){
  if(++(t -> size) > t -> capacity){
    tree_reallocate(t);
  }
  t -> contents[t -> size - 1] = data;
}

void tree_remove_node(tree* t){
  if(t -> size > 0){
    (t -> size)--;
  }
}

void tree_destroy(tree* t){
  free(t -> contents);
}

int tree_root(tree* t){
  return t -> contents[0];
}

int tree_left_child_index(tree* t, int parent_index){
  return 2 * (parent_index + 1) - 1;
}

int tree_right_child_index(tree* t, int parent_index){
  return 2 * (parent_index + 1);
}

int tree_left_child(tree* t, int parent_index){
  return t -> contents[tree_left_child_index(t, parent_index)];
}

int tree_right_child(tree* t, int parent_index){
  return t -> contents[tree_right_child_index(t, parent_index)];
}

int main(){
  tree _t;
  tree* t = &_t;

  tree_initialize(t);
  tree_add_node(t, 1);
  tree_add_node(t, 2);
```

```
  tree_add_node(t, 3);
  tree_add_node(t, 4);
  tree_add_node(t, 5);
  tree_add_node(t, 6);

  printf("Tree has %d nodes.\n", t -> size);

  for(int i = 0; i < t -> size; i++){
    printf("%d: %d\n", i, t -> contents[i]);
  }

  printf("Root: %d\n", tree_root(t));
  printf("Left Child: %d\n", tree_left_child(t, 0));
  printf("Right Child: %d\n", tree_right_child(t, 0));
  printf("Left Child: %d\n", tree_left_child(t, tree_left_child_index(t, 0)));
  printf("Right Child: %d\n", tree_right_child(t, tree_left_child_index(t, 0)));

  return 0;
}
```

./tree/tree-binary-complete/includes/tree-binary-complete.h

```c
#ifndef COMPLETE_BINARY_TREE_H
#define COMPLETE_BINARY_TREE_H

typedef struct tree_binary_complete tree;

struct tree_binary_complete {
  int size;
  int capacity;
  int* contents;
};

void tree_initialize(tree* t);
void tree_reallocate(tree* t);
void tree_add_node(tree* t, int data);
void tree_remove_node(tree* t);
void tree_destroy(tree* t);
int tree_left_child(tree* t, int parent_index);
int tree_right_child(tree* t, int parent_index);

#endif
```

./tree/tree-generic/tree_generic.c

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node node;
typedef struct nodelist nodelist;
typedef struct tree_generic tree;

struct node {
  int data;
  nodelist* children;
};

struct nodelist {
  int size;
  int capacity;
  node** contents;
};

struct tree_generic {
  node* root;
};

node* node_create_new(int data){
  node* temp = malloc(sizeof(node));
  temp -> data = data;
  temp -> children = NULL;
  return temp;
}

void nodelist_initialize(nodelist* v){
  v -> size = 0;
  v -> capacity = 0;
  v -> contents = NULL;
}

void nodelist_reallocate(nodelist* v){
  if(v -> capacity == 0){
    v -> contents = malloc(sizeof(node*));
    v -> capacity = 1;
  } else {
    v -> contents = realloc(v -> contents, sizeof(node*) * ( v -> capacity * 2 ));
    v -> capacity *= 2;
  }
}

void nodelist_push_back(nodelist* v, int data){
  if(++(v -> size) > v -> capacity){
    nodelist_reallocate(v);
  }
  v -> contents[v -> size - 1] = node_create_new(data);
}

void nodelist_pop_back(nodelist* v){
  if(v -> size > 0){
    (v -> size)--;
  }
}

void tree_initialize(tree *t){
  t -> root = NULL;
}

void tree_add_child(node* parent, int data){
  if(parent -> children == NULL){
```

```
    parent -> children = malloc(sizeof(nodelist));
  }
  nodelist_push_back(parent -> children, data);
}

node* tree_get_child(node* parent, int index){
  return parent -> children -> contents[index];
}

void indent(int levels, int ends, int flags){
  char* indentation[levels];
  for(int i = levels - 1; i >= 0; i--){
    if(i == levels - 1){
      if(ends) indentation[i] = "└─";
      else indentation[i] = "├─";
    } else {
      if(flags % 2 == 0) indentation[i] = "  ";
      else indentation[i] = "│ ";
    }
    flags = (flags >> 1);
  }
  for(int i = 0; i < levels; i++){
    printf("%s", indentation[i]);
  }
}

void traverse(node* n, int levels, int flags){
  printf("%d\n", n -> data);
  if(n -> children){
    for(int i = 0; i < n -> children -> size; i++){
      int ends = i == n -> children -> size - 1;
      flags = (flags << 1);
      if(!ends) flags += 1;
      indent(levels + 1, ends, flags);
      traverse(n -> children -> contents[i], levels + 1, flags);
      flags = (flags >> 1);
    }
  }
}

void tree_destroy(node* root){
  for(int i = 0; i < root -> children -> size; i++){
    free(root -> children -> contents[i]);
  }
  free(root -> children);
  free(root);
}

int main(){

  tree _t;
  tree* t = &_t;

  tree_initialize(t);

  t -> root = node_create_new(1);
  node* root = t -> root;

  tree_add_child(root, 2);
  node* child0 = tree_get_child(root, 0);
  tree_add_child(child0, 6);
  tree_add_child(child0, 7);
  node* child01 = tree_get_child(child0, 1);
  tree_add_child(child01, 12);
```

```
  tree_add_child(root, 3);
  node* child1 = tree_get_child(root, 1);
  tree_add_child(child1, 8);
  node* child10 = tree_get_child(child1, 0);
  tree_add_child(child10, 13);
  tree_add_child(child10, 14);

  tree_add_child(root, 4);
  node* child2 = tree_get_child(root, 2);
  tree_add_child(child2, 9);
  tree_add_child(child2, 10);
  tree_add_child(child2, 11);

  tree_add_child(root, 5);
  node* child3 = tree_get_child(root, 3);
  tree_add_child(child3, 12);
  node* child30 = tree_get_child(child3, 0);
  tree_add_child(child30, 13);
  node* child300 = tree_get_child(child30, 0);
  tree_add_child(child300, 14);
  tree_add_child(child30, 15);
  tree_add_child(child30, 16);
  node* child302 = tree_get_child(child30, 2);
  tree_add_child(child302, 17);

  traverse(root, 0, 0);

  tree_destroy(root);

  nodelist _n;
  nodelist* n = &_n;

  return 0;
}
```

./tree/heap/heap.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "includes/heap.h"

void heap_max_initialize(heap* h){
  h -> size = 0;
  h -> capacity = 0;
  h -> max = 1;
  h -> contents = NULL;
}

void heap_min_initialize(heap* h){
  h -> size = 0;
  h -> capacity = 0;
  h -> max = 0;
  h -> contents = NULL;
}

void heap_swap(heap* h, int x, int y){
  int temp = h -> contents[x];
  h -> contents[x] = h -> contents[y];
  h -> contents[y] = temp;
}

int heap_parent(int child_index){
  // (for 1-based indexing) parent of i is i / 2
  // (for 0-based indexing) parent of i is (i + 1) / 2 - 1
  return (child_index + 1) / 2 - 1;
}

int heap_left_child(heap* h, int parent_index){
  // (for 1-based indexing) left child of i is 2 * i
  // (for 0-based indexing) left child of i is 2 * (i + 1) - 1
  return 2 * (parent_index + 1) - 1;
}

int heap_right_child(heap* h, int parent_index){
  // (for 1-based indexing) right child of i is 2 * i + 1
  // (for 0-based indexing) right child of i is 2 * (i + 1) + 1 - 1
  return 2 * (parent_index + 1);
}

void heap_reallocate(heap* h){
  if(h -> capacity == 0){
    h -> contents = (int*) malloc(sizeof(int));
    h -> capacity = 1;
  } else {
    h -> contents = (int*) realloc(h -> contents, sizeof(int) * ( h -> capacity * 2 ));
    h -> capacity *= 2;
  }
}

void heap_heapify_insertion(heap *h){
  int i = h -> size - 1;
  while(i > 0){
    int parent = heap_parent(i);
    int check = h -> contents[parent] < h -> contents[i];
    if(!h -> max) check = !check;
    if(check){
      heap_swap(h, parent, i);
      i = parent;
    } else return;
  }
}
```

./tree/heap/heap.c

```
void heap_push(heap* h, int data){
  if(++(h -> size) > h -> capacity){
    heap_reallocate(h);
  }
  h -> contents[h -> size - 1] = data;
  heap_heapify_insertion(h);
}

void heap_heapify_deletion(heap* h){
    int i = 0;
    while((heap_left_child(h, i)) < h -> size){
      int left = heap_left_child(h, i);
      int right = heap_right_child(h, i);
      int prior = left;
      if(right < h -> size){
        if(h -> max) prior = h -> contents[left] > h -> contents[right] ? left : right;
        else prior = h -> contents[left] < h -> contents[right] ? left : right;
      }
      int check = h -> contents[prior] > h -> contents[i];
      if(!h -> max) check = !check;
      if(check){
        heap_swap(h, prior, i);
        i = prior;
      } else return;
    }

}

void heap_pop(heap* h){
  if(h -> size > 0){
    h -> contents[0] = h -> contents[h -> size - 1];
    (h -> size)--;
    heap_heapify_deletion(h);
  }
}

int heap_top(heap* h){
  return h -> contents[0];
}

void heap_destroy(heap* h){
  free(h -> contents);
}

void heap_display(heap* h){
  for(int i = 0; i < h -> size; i++){
    printf("%d ", h -> contents[i]);
  } printf("\n");
}
```

./tree/heap/makefile

```makefile
CC = gcc
NAME = heap
FILES = main.c heap.c

compile:
	$(CC) -o $(NAME) $(FILES)

run: compile
	./$(NAME)

debug:
	$(CC) -o $(NAME) -g $(FILES)

clean:
	rm ${NAME}
```

./tree/heap/main.c

```c
#include "includes/heap.h"

int main(){
  heap _h;
  heap* h = &_h;

  heap_max_initialize(h);

  heap_push(h, 70);
  heap_push(h, 40);
  heap_push(h, 45);
  heap_push(h, 50);
  heap_push(h, 30);
  heap_push(h, 20);
  heap_push(h, 10);

  heap_display(h);

  heap_pop(h);
  heap_display(h);

  heap_pop(h);
  heap_display(h);

  heap_pop(h);
  heap_display(h);

  heap_pop(h);
  heap_display(h);

  heap_pop(h);
  heap_display(h);

  heap_push(h, 500);
  heap_push(h, 20);
  heap_push(h, 50);

  heap_display(h);

}
```

./tree/heap/includes/heap.h

```
#ifndef HEAP_H
#define HEAP_H

typedef struct priority_queue heap;

struct priority_queue {
  int size;
  int capacity;
  char max;
  // max = 1 for maximum priority queue
  // max = 0 for minimum priority queue
  int* contents;
};

void heap_max_initialize(heap* h);
void heap_min_initialize(heap* h);
int heap_parent(int child_index);
int heap_left_child(heap* h, int parent_index);
int heap_right_child(heap* h, int parent_index);
void heap_reallocate(heap* h);
void heap_push(heap* h, int data);
void heap_pop(heap* h);
int heap_top(heap* h);
void heap_destroy(heap* h);
void heap_display(heap* h);

#endif
```

./dynamic-array/string.c

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
  int size;
  int capacity;
  char* contents;
} string;

void string_initialize(string* v){
  v -> size = 0;
  v -> capacity = 0;
  v -> contents = NULL;
}

void string_reallocate(string* v){
  if(v -> capacity == 0){
    v -> contents = malloc(sizeof(char));
    v -> capacity = 1;
  } else {
    v -> contents = realloc(v -> contents, sizeof(char) * ( v -> capacity * 2 ));
    v -> capacity *= 2;
  }
}

void string_push_back(string* v, char data){
  if(++(v -> size) > v -> capacity){
    string_reallocate(v);
  }
  v -> contents[v -> size - 1] = data;
}

void string_pop_back(string* v){
  if(v -> size > 0){
    (v -> size)--;
  }
}

char string_at(string* v, int index){
  return v -> contents[index];
}

void string_set(string* s, char* data, int size){
  string_initialize(s);
  for(int i = 0; i < size; i++){
    string_push_back(s, data[i]);
  }
}

void string_concatenate(string* s, char* data, int size){
  for(int i = 0; i < size; i++){
    string_push_back(s, data[i]);
  }
}

void string_destroy(string* v){
  free(v -> contents);
}

int main(){
  string _s;
  string* s = &_s;

  string_initialize(s);
```

./dynamic-array/string.c

```
  string_push_back(s, 's');
  string_push_back(s, 'u');
  string_push_back(s, 'y');
  string_push_back(s, 'a');
  string_push_back(s, 's');
  printf("%s\n", s -> contents);
  printf("size: %d\n", s -> size);

  string_set(s, "nomore", 6);
  printf("%s\n", s -> contents);
  printf("size: %d\n", s -> size);

  string_concatenate(s, " is more", 8);
  printf("%s\n", s -> contents);
  printf("size: %d\n", s -> size);

  string_destroy(s);

  return 0;
}
```

./dynamic-array/dynamic-array.c

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct dynamic_array vector;

struct dynamic_array {
  int size;
  int capacity;
  int* content;
};

void vector_initialize(vector* v){
  v -> size = 0;
  v -> capacity = 0;
  v -> content = NULL;
}

void vector_reallocate(vector* v){
  if(v -> capacity == 0){
    v -> content = malloc(sizeof(int));
    v -> capacity = 1;
  } else {
    v -> content = realloc(v -> content, sizeof(int) * ( v -> capacity * 2 ));
    v -> capacity *= 2;
  }
}

void vector_push_back(vector* v, int data){
  if(++(v -> size) > v -> capacity){
    vector_reallocate(v);
  }
  v -> content[v -> size - 1] = data;
}

void vector_pop_back(vector* v){
  if(v -> size > 0){
    (v -> size)--;
  }
}

int vector_at(vector* v, int index){
  return v -> content[index];
}

void vector_destroy(vector* v){
  free(v -> content);
}

int main(){
  vector _v;
  vector* v = &_v;

  vector_initialize(v);
  for(int i = 0; i < 10; i++){
    vector_push_back(v, i);
  }

  for(int i = 0; i < v -> size; i++){
    printf("%d ", vector_at(v, i));
  } printf("\n");

  printf("size: %d capacity: %d", v -> size, v -> capacity);
  vector_destroy(v);
  return 0;
```

```
}
```

./graph/makefile

```
CC = gcc
NAME = graph
FILES = main.c graph.c

compile:
  $(CC) -o $(NAME) $(FILES)

run: compile
  ./$(NAME)

debug:
  $(CC) -o $(NAME) -g $(FILES)

clean:
  rm ${NAME}
```

./graph/main.c

```c
#include "./includes/graph.h"

int main(){

  const int input[][3] = {
    {1, 2, 28},
    {1, 6, 10},
    {2, 3, 16},
    {2, 7, 14},
    {3, 4, 12},
    {4, 5, 22},
    {4, 7, 18},
    {5, 6, 25},
    {5, 7, 24},
  };

  graph _g;
  graph* g = &_g;
  int vertices = 7, edges = 9;
  graph_initialize(g, vertices);

  graph_populate_from_input(g, edges, input);

  graph_print_adjacency_list_repr(g, vertices);

  graph_destroy(g, vertices);

  return 0;
}
```

./graph/graph.c

```
#include <stdio.h>
#include <stdlib.h>
#include "includes/graph.h"

// Increases capacity of dynamic array `edgelist`
// when edgelist needs to store after fully filled.
void edgelist_reallocate(edgelist* e){
  if(e -> capacity == 0){
    e -> contents = malloc(sizeof(int));
    e -> capacity = 1;
  } else {
    e -> contents = realloc(e -> contents, sizeof(int) * ( e -> capacity * 2 ));
    e -> capacity *= 2;
  }
}

// Pushes a new `edge` initialized from `vertex` 'source'
// to `vertex` 'destination' weighted 'weight'
// at the end of `edgelist` 'e'
void edgelist_push_back(edgelist* e, vertex* source, vertex* destination, int weight){
  if(++(e -> size) > e -> capacity){
    edgelist_reallocate(e);
  }
  e -> contents[e -> size - 1] = edge_new(source, destination, weight);
}

// Removes the last `edge` from the `edgelist` 'e'
void edgelist_pop_back(edgelist* e){
  if(e -> size > 0){
    (e -> size)--;
  }
}

edge* edge_new(vertex* source, vertex* destination, int weight){
  edge* temp = malloc(sizeof(edge));
  temp -> source = source;
  temp -> destination = destination;
  temp -> weight = weight;
  return temp;
}

// Initializes a `graph` 'g' with 'n' vertices
void graph_initialize(graph* g, int n){
  g -> adjacency = malloc(sizeof(vertex) * n);
  for(int i = 0; i < n; i++){
    vertex* vi = &(g -> adjacency[i]);
    vi -> index = i + 1;
    vi -> edges = (edgelist*) malloc(sizeof(edgelist));
    vi -> edges -> size = 0;
    vi -> edges -> capacity = 0;
    vi -> edges -> contents = NULL;
  }
}

void graph_destroy(graph* g, int n){
  for(int i = 0; i < n; i++){
    vertex* vi = &(g -> adjacency[i]);
    free(vi -> edges);
  }
  free(g -> adjacency);
}

void graph_print_adjacency_list_repr(graph* g, int vertices){
  for(int i = 0; i < vertices; i++){
```

```
    vertex* vi = &(g -> adjacency[i]);
    int ei = vi -> edges -> size;
    if(ei){
      printf("%d: [", i + 1);
      for(int j = 0; j < ei; j++){
        printf("%d, ", vi -> edges -> contents[j] -> source -> index);
        printf("%d, ", vi -> edges -> contents[j] -> destination -> index);
        printf("%d", vi -> edges -> contents[j] -> weight);
        printf("]");
        if(j != ei - 1) printf(", [");
      } printf("\n");
    }
  }
}

void graph_populate_from_input(graph* g, int nedges, const int input[][3]){
  for(int i = 0; i < nedges; i++){
    vertex* source = &(g -> adjacency[input[i][0] - 1]);
    vertex* destination = &(g -> adjacency[input[i][1] - 1]);
    edgelist_push_back(source -> edges, source, destination, input[i][2]);
    edgelist_push_back(destination -> edges, destination, source, input[i][2]);
  }
}
```

./graph/algorithms/depth-first-search/makefile

```
CC = gcc
NAME = dfs
FILES = dfs.c ../../graph.c

compile:
  $(CC) -o $(NAME) $(FILES)

run: compile
  ./$(NAME)

debug:
  $(CC) -o $(NAME) -g $(FILES)

clean:
  rm ${NAME}
```

./graph/algorithms/depth-first-search/dfs.c

```c
#include <stdio.h>
#include "../../includes/graph.h"

void DFS(graph* g, int nvertex, int src, int* visited){
  visited[src] = 1;
  printf("%d ", src + 1);
  for(int i = 0; i < g -> adjacency[src].edges -> size; i++){
    int e = g -> adjacency[src].edges -> contents[i] -> destination -> index;
    if(!visited[e - 1]){
      DFS(g, nvertex, e - 1, visited);
    }
  }
}

int main(){
  // User Input as a variable
  const int input[][3] = {
    {1, 2, 28},
    {1, 6, 10},
    {2, 3, 16},
    {2, 7, 14},
    {3, 4, 12},
    {4, 5, 22},
    {4, 7, 18},
    {5, 6, 25},
    {5, 7, 24},
  };

  graph _g;
  graph* g = &_g;

  int vertices = 7, edges = 9;
  graph_initialize(g, vertices);
  graph_populate_from_input(g, edges, input);

  // graph_print_adjacency_list_repr(g ,vertices);

  int visited[vertices];
  for(int i = 0; i < vertices; i++) visited[i] = 0;
  printf("DFS sequence: ");
  DFS(g, vertices, 0, visited);
  printf("\n");

  graph_destroy(g, vertices);
}
```

./graph/algorithms/breadth-first-search/makefile

```
CC = gcc
NAME = bfs
FILES = bfs.c ../../graph.c ../../../queue/queue.c

compile:
  $(CC) -o $(NAME) $(FILES)

run: compile
  ./$(NAME)

debug:
  $(CC) -o $(NAME) -g $(FILES)

clean:
  rm ${NAME}
```

./graph/algorithms/breadth-first-search/bfs.c

```c
#include "../../includes/graph.h"
#include "../../../queue/includes/queue.h"
#include <stdio.h>

void BFS(graph* g, int nvertex, int src, int* visited){
  // creating queue
  queue _q;
  queue* q = &_q;
  queue_initialize(q);

  // add src to queue
  queue_enqueue(q, src + 1);
  while(!queue_empty(q)){
    int c = queue_front(q);
    queue_dequeue(q);
    if(!visited[c - 1]){
      visited[c - 1] = 1;
      printf("%d ", c);
      for(int i = 0; i < g -> adjacency[c - 1].edges -> size; i++){
        int e = g -> adjacency[c - 1].edges -> contents[i] -> destination -> index;
        if(!visited[e - 1]){
          queue_enqueue(q, e);
        }
      }
    }
  }
}

int main(){
  // User Input as a variable
  const int input[][3] = {
    {1, 2, 28},
    {1, 6, 10},
    {2, 3, 16},
    {2, 7, 14},
    {3, 4, 12},
    {4, 5, 22},
    {4, 7, 18},
    {5, 6, 25},
    {5, 7, 24},
  };

  graph _g;
  graph* g = &_g;

  int vertices = 7, edges = 9;
  graph_initialize(g, vertices);
  graph_populate_from_input(g, edges, input);

  // graph_print_adjacency_list_repr(g ,vertices);

  int visited[vertices];
  for(int i = 0; i < vertices; i++) visited[i] = 0;
  printf("BFS sequence: ");
  BFS(g, vertices, 0, visited);
  printf("\n");

  graph_destroy(g, vertices);
}
```

./graph/algorithms/prims/makefile

```
CC = gcc
NAME = prims
FILES = prims.c ../../graph.c

compile:
  $(CC) -o $(NAME) $(FILES)

run: compile
  ./$(NAME)

debug:
  $(CC) -o $(NAME) -g $(FILES)

clean:
  rm ${NAME}
```

./graph/algorithms/prims/prims.c

```
#include "../../includes/graph.h"

// prim's minimum spanning tree algorithm
void graph_prims_mst(graph* g, int vertices){
  int visits[vertices];
  for(int i = 0; i < vertices; i++) visits[i] = 0;
}

int main(){

  // User Input as a variable
  const int input[][3] = {
    {1, 2, 28},
    {1, 6, 10},
    {2, 3, 16},
    {2, 7, 14},
    {3, 4, 12},
    {4, 5, 22},
    {4, 7, 18},
    {5, 6, 25},
    {5, 7, 24},
  };

  graph _g;
  graph* g = &_g;

  int vertices = 7, edges = 9;
  graph_initialize(g, vertices);
  graph_populate_from_input(g, edges, input);

  graph_destroy(g, vertices);

  return 0;
}
```

./graph/includes/graph.h

```c
#ifndef GRAPH_H
#define GRAPH_H

typedef struct edge edge;
typedef struct edgelist edgelist;
typedef struct vertex vertex;
typedef struct graph graph;

struct edgelist {
  int size;
  int capacity;
  edge** contents;
};

struct vertex {
  int index;
  edgelist* edges;
};

struct edge {
  vertex* source;
  vertex* destination;
  int weight;
};

struct graph{
  vertex* adjacency;
};

void edgelist_reallocate(edgelist* e);
void edgelist_push_back(edgelist* e, vertex* source, vertex* destination, int weight);
void edgelist_pop_back(edgelist* e);
edge* edge_new(vertex* source, vertex* destination, int weight);
void graph_initialize(graph* g, int n);
void graph_destroy(graph* g, int n);
void graph_print_adjacency_list_repr(graph* g, int vertices);
void graph_populate_from_input(graph* g, int nedges, const int input[][3]);

#endif
```

./queue/queue.c

```c
#include <stdlib.h>
#include "includes/queue.h"

void queue_initialize(queue* q){
  q -> front = NULL;
  q -> back  = NULL;
}

void queue_enqueue(queue* q, int data){

  // initializing new node
  node* temp = (node *) malloc(sizeof(node));
  temp -> data = data;
  temp -> next = NULL;
  temp -> prev = q -> back;

  // if not last node , modify next to previous node
  if(q -> back) q -> back -> next = temp;

  // if first node, handle front
  if(q -> front == NULL) q -> front = temp;

  // inserting node at back
  q -> back = temp;
}

void queue_dequeue(queue* q){
  if(q -> front){
    node* temp = q -> front;
    q -> front = temp -> next;
    if(q -> front) q -> front -> prev = NULL;
    free(temp);
  }
}

int queue_front(queue* q){
  return q -> front -> data;
}

int queue_empty(queue* q){
  if(q -> front) return 0;
  return 1;
}
```

./queue/main.c

```
#include "includes/queue.h"
#include <stdio.h>

void queue_front_print(queue *q){
  if(q -> front){
    printf("%d\n", queue_front(q));
  } else {
    printf("Queue Empty!\n");
  }
}

int main(){
  queue _q;
  queue* q = &_q;
  queue_initialize(q);
  queue_enqueue(q, 1);
  queue_enqueue(q, 2);
  queue_enqueue(q, 3);

  queue_front_print(q);
  queue_dequeue(q);
  queue_front_print(q);
  queue_dequeue(q);
  queue_front_print(q);
  queue_dequeue(q);
  queue_front_print(q);
  return 0;
}
```

./queue/includes/queue.h

```
#ifndef QUEUE_H
#define QUEUE_H

typedef struct node node;

struct node {
  int data;
  node* next;
  node* prev;
};

typedef struct queue {
  node* front;
  node* back;
} queue;


void queue_initialize(queue* q);
void queue_enqueue(queue* q, int data);
void queue_dequeue(queue* q);
int queue_front(queue* q);
void queue_print(queue* q);
int queue_empty(queue* q);


#endif
```

./stack/stack.c

```
#include <stdio.h>
#include <stdlib.h>

typedef struct node node;

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

/*
 * LINKED LIST IMPLEMENTATION OF STACK <int>
 * */
typedef struct stack {
  node* top;
} stack;

void stack_initialize(stack* s);
void stack_push(stack* s, int data);
void stack_pop(stack* s);
node* stack_top(stack* s);
void stack_top_print(stack *s);
void stack_print(stack* s);

// Initializes an empty stack
void stack_initialize(stack* s){
  s -> top = NULL;
}

// pushes single integer to top of stack
void stack_push(stack* s, int data){
  node* temp = (node *) malloc(sizeof(node));
  temp -> data = data;
  temp -> next = s -> top;
  s -> top = temp;
}

// pops single integer from top of stack
void stack_pop(stack* s){
  if(s -> top){
    node* temp = s -> top;
    s -> top = s -> top -> next;
    free(temp);
  }
}

// returns node at top of stack
node* stack_top(stack* s){
  return s -> top;
}

// prints integer at top of stack
void stack_top_print(stack *s){
  if(s -> top){
    printf("%d\n", s -> top -> data);
  } else {
    printf("Stack Empty!");
  }
}

void stack_print(stack* s){
  while(s -> top != NULL){
    printf("%d ", s -> top -> data);
    stack_pop(s);
```

```
  }
}

int main(){
  stack s_t;
  stack *s = &s_t;
  s -> top = NULL;
  stack_push(s, 5);
  stack_push(s, 4);
  stack_push(s, 3);
  stack_push(s, 2);
  stack_push(s, 1);
  stack_push(s, 0);
  stack_top_print(s);
  stack_pop(s);
  stack_top_print(s);
  stack_pop(s);
  stack_top_print(s);
  stack_pop(s);
  stack_top_print(s);
  stack_pop(s);
  stack_top_print(s);
  stack_pop(s);
  stack_top_print(s);
  stack_pop(s);
  stack_top_print(s);
  return 0;
}
```