RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Assignment

Course Code	CSC209A
Course Name	Design and Analysis of Algorithms
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Satyajit Ghana
Reg. No.	17ETCS002159
Semester/Year	04/2017
Course Leader(s)	Pallavi R Kumar

Declaration Sheet

Student Name	Satyajit Ghana		
Reg. No	17ETCS002159		
Programme	B.Tech	Semester/Year	04/2017
Course Code	CSC209A		
Course Title	Design and Analysis of Algorithms		
Course Date		to	
Course Leader	Pallavi R Kumar		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

Contents

List of Figures

1 Question 1

Solution to Question No. 1 Part A

1.1 Introduction to amortized analysis

In an *amortized analysis*, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized *analysis guarantees the average performance of each operation in the worst case.*

Some of the most commonly used techniques in Amortized analysis are:

Aggregate Analysis: In which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations.

Accounting Method: In which we determine an amortized cost of each operation

Potential Method: This is like the accounting method; this method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

1.2 Example illustrating amortized analysis

In the consecutive example we will be using STACK operations to illustrate one of the techniques used in amortized analysis which is ***Potential Method***.

PUSH(S, x) pushed object x onto stack S.

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2     POP(S)
3     k = k − 1
```

The potential method works as follows. We perform n operations on the initial Data Structure D_0. For each $i = 1, 2, \ldots, n$ we let c_i be the actual cost of the ith operation and D_i

be the result after the ith operation to the Data Structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to real number $\Phi(D_i)$, which is the ***potential*** associated with data structure D_i. The ***amortized cost*** c_i of the ith operation w.r.t the potential function Φ is defined by

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The amortized cost of the n operations is,

$$\begin{aligned}\sum_{i=1}^{n} c_i = & \sum_{i=1}^{n}\big(c_i + \Phi(D_i) - \Phi(D_{i-1})\big) \\ = & \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack, we have $\Phi(D_0) = 0$. The stack D_i after the ith operation has nonnegative potential thus,

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0)\end{aligned}$$

Let us now compute the amortized cost of various stack operations. If the ith operation on the stack containing s objects is a PUSH operation, then the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1\end{aligned}$$

The amortized cost of this PUSH operation is

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 = 2\end{aligned}$$

Suppose the ith operation on the stack is MULTIPOP(S, k) which causes k'=min(k, s) objects to be popped off the stack. The actual cost of the operation is k', the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' = 0\end{aligned}$$

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$.

(CLRS, Introduction to Algorithms, pp 451-459, Amortized Analysis)

1.3 Conclusion

As the debate goes by "Pessimistic approach in Amortized Analysis", The worst case running time might give an overly pessimistic analysis for algorithms performing a sequence of operations on data structures. In amortized analysis, we average the time required to perform a sequence of operations over the number of operations performed.

Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can "charge them" to the cheap operations, and only bound the cheap operations.

2 Question 2

Solution to Question 1 Part B

2.1 Algorithm

To solve this problem we use Prim's Algorithm, the basic idea behind being that the minimum spanning tree must be created such that all the vertices be connected with the minimum weight edge.

Prim's Algorithm

1. Create a mst_set that keeps track of vertices already included in MST
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first
3. while mst_set doesn't include all vertices
    a. pick a vertex u which is not there in mst_set and has minimum key value
    b. include u to mst_set
    c. update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v.

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

The Original Algorithm given by RC Prim

```
1. T = $\phi$
2. U = $\{1\}$
3. while $(U \neq V)$
4.     let $(u, v)$ be the lowest cost edge such that $u \in U$ and $v \in V - U$
5. T = $T \cup \{(u, v)\}$
6. U = $U \cup \{v\}$
```

(RC Prim, Bell System Technical Journal, 1957)

The Algorithm used in the Program

MST-PRIM(G, w, r)

```
1.  for each u $\in$ G, V
2.      u.key = $\infty$
3.      u.$\pi$ = NIL
4.  r.key = 0
5.  Q = G.V
6.  while Q $\neq \emptyset$
7.      u = extract-min(Q)
8.      for each v $\in$ G.Adj[u]
9.          if v $\in$ Q and w(u, v) < v.key
10.             v.$\pi$ = u
11.             v.key = w(u, v)
```

(CLRS, Introduction to Algorithms)

2.2 A C Program

```c
int extract_min(int V, int *key_val, bool *mst_set) {
    int min = INT_MAX;
    int min_index = 0;

    for (int v = 0 ; v < V ; v++) {
        if (mst_set[v] == false && key_val[v] < min) {
            min = key_val[v], min_index = v;
        }
    }

    return min_index;
}

void find_cost(int V, int graph[][MAX]) {
    int parent[V], key_val[V];
    bool mst_set[V];

    for (int i = 0 ; i < V ; i++) {
        key_val[i] = INT_MAX, mst_set[i] = false;
    }

    parent[0] = -1;
    key_val[0] = 0;
    for (int count = 0 ; count < V - 1 ; count++) {
        int u = extract_min(V, key_val, mst_set);
        mst_set[u] = true;

        for (int v = 0 ; v < V ; v++) {
            if (graph[u][v] && mst_set[v] == false &&
                graph[u][v] < key_val[v]) {
                key_val[v] = graph[u][v];
                parent[v] = u;
            }
        }
    }

    int cost = 0;

    for (int v = 1 ; v < V ; v++)
        cost += graph[parent[v]][v];

    printf("minimum cost to reconnect all the computers : %d\n", cost);
}

int print_mst(int parent[], int V, int graph[][MAX]) {
    printf("Edge\tWeight\n");
    for (int v = 1; v < V; v++)
        printf("%d - %d \t%d \n", parent[v], v, graph[v][parent[v]]);
}
```

2.3 Analysis of Time and Space complexity

The outer loop(i.e. the loop to add new node to MST) runs n times and in each iteration of the loop it takes O(n) time to find the minnode and O(n) time to update the neighboring nodes of u-th node. Hence the overall complexity is O(n^2). Although this can be improved by using a Priority Queue to find the min-node, using which the time complexity for searching the min-node in the heap would be $O(\log n)$. Hence the total time complexity would be $O(E \log V)$ with the help of binary heap which is $O(n \log n)$.

When it comes to the Space Complexity, since we are storing the graph in form of an adjacency matrix, we would require a maximum of $O(EV)$ space, but if we were to consider the upper bound then the max value of edges can be that of the vertices. Hence the total space complexity would be $O(n^2)$

3 Question 3

Solution to Question 2 Part B

3.1 Design of the data structure QUACK

source **quack.h**

```c
struct Quack {
    Stack *s1, *s2;
    void (*push)(struct Quack*, void*);
    void* (*pop)(struct Quack*);
    void* (*pull)(struct Quack*);
    void (*display)(struct Quack*, void (*)(void*));
};

typedef struct Quack Quack;

/* To Initialize the Quack Structure */
Quack* newQuack(void);
void init_quack(Quack*);

/* Quack Operations */
void enqueue_quack(Quack*, void*);
void* dequeue_quack(Quack*);
void* pop_quack(Quack*);

/* print utility */
void print_quack(Quack*, void (*)(void*));
```

Basically the QUACK structure consists of two stacks s1 and s2 for storing the data, further we will elaborate on why this is so, and four functions, push to add data to the structure, pop and pull to remove data and display to print the data. The way the structure is designed, it is highly generic and can store any form of data, without any size limitations.

The data is stored in stack s1, the other stack s2 is used as a temporary place to store the data when various operations are being performed. Push and Pull operations are basically Enqueue and Dequeue operations on a Queue, hence we design the structure to perform like a queue using these two stacks. Whereas Pop is a stack pop operation from the stack.

The time complexity for the above mentioned operations are mentioned in 3.2.

3.2 Algorithms for operations on stack and queue

STACK-PUSH(data)

```
1. top = top + 1
2. stack[top] = data
```

Time Complexity : $O(1)$

STACK-POP()

```
1. if top == -1 display "ERROR: STACK UNDERFLOW"
2.    return NULL
3. data = stack[top]
4. top = top - 1
5. return data
```

Time Complexity : $O(1)$

STACK-DISPLAY()

```
1. for i = top to 0
2.    display stack[i]
```

Time Complexity : $O(n)$

ENQUEUE(data)

```
1. while(!s1.isEmpty())
2.   s2.push(s1.pop())
3. s1.push(data)
4. while(!s2.isEmpty())
5.   s1.push(s2.pop())
```

Time Complexity : $O(n)$

DEQUEUE()

```
1. if s1.isEmpty
2.    display "ERROR: QUACK is empty"
3.    return NULL
4. data = s1.pop()
5. return data
```

Time Complexity : $O(1)$

QUACK-POP()

```
1. if s1.isEmpty()
2.    display "ERROR: QUACK is empty"
3.    return NULL
4. while (!s1.isEmpty())
5.    s2.push(s1.pop())
6. data = s2.pop()
7. while (!s2.isEmpty())
8.    s1.push(s2.pop())
9. return data
```

Time Complexity : $O(n)$

DISPLAY-QUACK()

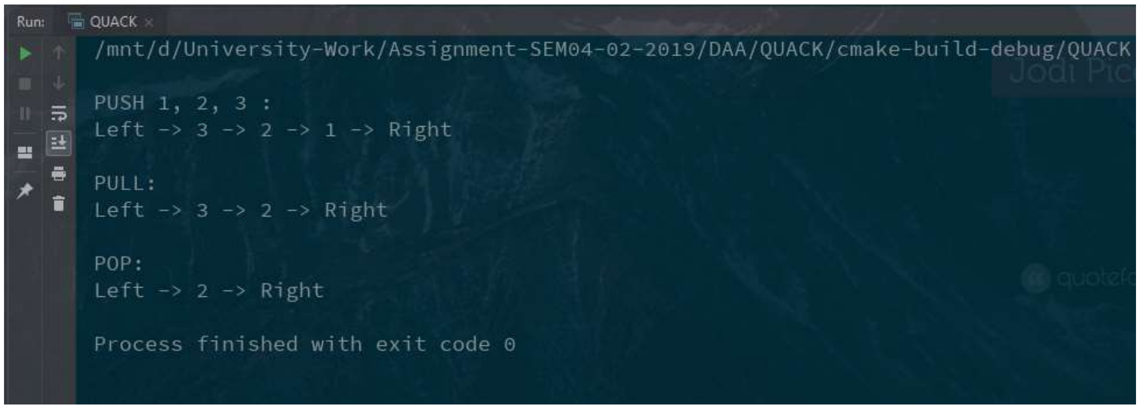
```
1. if s1.isEmpty()
2.    display "ERROR: QUACK is empty"
3.    return NULL
4. while (!s1.isEmpty())
5.    s2.push(s1.pop())
6. s2.display()
7. while (!s2.isEmpty())
8.    s1.push(s2.pop())
```

Time Complexity : $O(n)$

3.3 C functions for stack and queue operations

Refer to Appendix quack.c and quack.h for C source code

<u>OUTPUT:</u>



```
/mnt/d/University-Work/Assignment-SEM04-02-2019/DAA/QUACK/cmake-build-debug/QUACK

PUSH 1, 2, 3 :
Left -> 3 -> 2 -> 1 -> Right

PULL:
Left -> 3 -> 2 -> Right

POP:
Left -> 2 -> Right

Process finished with exit code 0
```

Bibliography

1. R.C. Prim. Shortest connection networks and some generalizations. Bell System Technical Journal, Volume 36, pp. 1389-1401, 1957.
2. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/
3. CLRS, Introduction to Algorithms, third edition, 2009.
4. http://www.cs.cornell.edu/courses/cs3110/2009sp/recitations/rec21.html

Appendix

QUACK Data Structure complete “pure C99 std” C source code

WARNING: Contains a lot of memory leaks, a lot!, Valgrind was too far fetched for such a problem + strict memory management might complicate the code.

quack.c

```c
//
// Created by shadowleaf on 05-Mar-19.
//

#include <stdlib.h>
#include "quack.h"
#include "limits.h"

/* Initialize the Quack Structure */
Quack* newQuack(void) {
    Quack* myquack;
    // Make sure the allocation was successful
    if ((myquack = malloc(sizeof *myquack)) != NULL) {
        init_quack(myquack);
        return myquack;
    }

    return NULL;
}

void init_quack(Quack* myquack) {
    // Assuming my quack structure can handle a huge amount
    // of data, lol.

    myquack -> s1 = newStack(INT_MAX);
    myquack -> s2 = newStack(INT_MAX);
    //myquack -> enqueue = enqueue_quack;
    //myquack -> dequeue = dequeue_quack;
    myquack -> push = enqueue_quack;
    myquack -> pop = pop_quack;
    myquack -> pull = dequeue_quack;
    myquack -> display = print_quack;
}

void print_quack(Quack* myquack, void (*print_one)(void*)) {
    // Get a nice little reference, meow
    Stack* s1 = myquack -> s1;
    Stack* s2 = myquack -> s2;

    while(!(s1 -> isEmpty(s1))) {
        s2 -> push(s2, s1 -> peek(s1));
        s1 -> pop(s1);
    }

    printf("Left -> ");
    s2 -> display (s2, print_one);
    printf("Right");

    while(!(s2 -> isEmpty(s2))) {
        s1 -> push(s1, s2 -> peek(s2));
        s2 -> pop(s2);
    }
}

void enqueue_quack(Quack* myquack, void* data) {
    // Get a nice little reference, meow
    Stack* s1 = myquack -> s1;
    Stack* s2 = myquack -> s2;

    while(!(s1 -> isEmpty(s1))) {
        s2 -> push(s2, s1 -> peek(s1));
        s1 -> pop(s1);
    }

    s1 -> push(s1, data);

    while(!(s2 -> isEmpty(s2))) {
        s1 -> push(s1, s2 -> peek(s2));
        s2 -> pop(s2);
    }
}

void* dequeue_quack(Quack* myquack) {
    Stack* s1 = myquack -> s1;
    Stack* s2 = myquack -> s2;

    if (s1 -> isEmpty(s1)) {
        printf("QUACK is Empty");
        return NULL;
    }

    void* x = s1 -> peek(s1);
    s1 -> pop(s1);

    return x;
}

void* pop_quack(Quack* myquack) {
    Stack* s1 = myquack -> s1;
    Stack* s2 = myquack -> s2;

    if (s1 -> isEmpty(s1)) {
        printf("QUACK is Empty");
        return NULL;
    }

    while(!(s1 -> isEmpty(s1))) {
        s2 -> push(s2, s1 -> peek(s1));
        s1 -> pop(s1);
    }

    void* x = s2 -> pop(s2);

    while(!(s2 -> isEmpty(s2))) {
        s1 -> push(s1, s2 -> peek(s2));
        s2 -> pop(s2);
    }

    return x;
}
```

quack.h

```
//
// Created by shadowleaf on 05-Mar-19.
//

#ifndef QUACK_QUACK_H
#define QUACK_QUACK_H

#include "stack.h"

struct Quack {
    Stack *s1, *s2;
    void (*push)(struct Quack*, void*);
    void* (*pop)(struct Quack*);
    void* (*pull)(struct Quack*);
    void (*display)(struct Quack*, void (*)(void*));
};

typedef struct Quack Quack;

/* To Initialize the Quack Structure */
Quack* newQuack(void);
void init_quack(Quack*);

/* Quack Operations */
void enqueue_quack(Quack*, void*);
void* dequeue_quack(Quack*);
void* pop_quack(Quack*);

/* print utility */
void print_quack(Quack*, void (*)(void*));

#endif //QUACK_QUACK_H
```

stack.c

```
//
// Created by shadowleaf on 05-Mar-19.
//
```

```c
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "input_helper.h"
#include "debug_helper.h"

Stack* newStack(int max_size) {
    Stack* mystack = malloc(sizeof *mystack);
    init_stack(mystack, max_size);
    return mystack;
}

void init_stack(Stack* mystack, int max_size) {
    mystack -> top = -1;
    mystack -> data = NULL;
    mystack -> MAX = max_size;
    mystack -> push = push;
    mystack -> pop = pop;
    mystack -> display = display;
    mystack -> peek = peek;
    mystack -> isEmpty = isEmpty;
}

void push(Stack* mystack, void* data) {

    if (mystack -> top >= mystack -> MAX) {
        printf("\n*Stack Overflow Detected !*\n");
        return;
    }

    /* This code sucks, you know it and i know it.
     * Move on and call me an idiot later */
    mystack -> data = realloc(mystack -> data, (mystack -> top + 2) *
sizeof *(mystack -> data));
    if (mystack -> data != NULL) {
        (mystack -> data)[(++mystack -> top)] = data;
    } else {
        printf("\n*cannot allocate memory !*\n");
    }
}

void* pop(Stack* mystack) {
    if (mystack -> top < 0) {
        printf("\n*Stack Underflow detected !*\n");
        return NULL;
    }

    mystack -> top--;
    /* Resize the data array as the elements are removed */
    mystack -> data = realloc(mystack -> data, (mystack -> top + 2) *
sizeof *(mystack-> data));

    return (mystack -> data)[(mystack -> top)+1];
}

void display(Stack* mystack) {
    for (int i = mystack -> top  ; i >= 0 ; i--) {
        printf("%s ", *(char**)(mystack->data + i));
    }
}

void* peek(Stack* mystack) {
    return (mystack -> data)[mystack -> top];
}

bool isEmpty(Stack* mystack) {
    if (mystack -> top < 0)
        return true;

    return false;
}
```

stack.h

```c
//
// Created by shadowleaf on 05-Mar-19.
//

#ifndef QUACK_STACK_H
#define QUACK_STACK_H

#include <stdio.h>
#include <stdbool.h>
#include "input_helper.h"

/* Why not use a Vector and delete first element, anyways
 * it adds elements to the end of the list, using Vectors method
 * there will be no size limitations */

struct Stack {
    int top;
    void **data;
    int MAX;
    void (*push)(struct Stack*, void*);
    void* (*pop)(struct Stack*);
    void (*display)(struct Stack*);
    void* (*peek)(struct Stack*);
    bool (*isEmpty)(struct Stack*);
};

typedef struct Stack Stack;

/* Initialize the Stack */
Stack* newStack(int);
void init_stack(Stack*, int);

/* Stack Operations */
void push(Stack*, void *);
void* pop(Stack*);
void display(Stack*);
void* peek(Stack*);
bool isEmpty(Stack*);
```

```
#endif //QUACK_STACK_H
```

vector.c

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "vector.h"
#include "debug_helper.h"

typedef int (*Comparator)(const void*, const void*);

Vector* newVector(void (*print_util)(Vector*),
        int (*compare)(const void*, const void*),
        int (*compare_r)(const void*, const void*)) {
    Vector* myVector = malloc(sizeof *myVector);
    init(myVector, print_util, compare, compare_r);
    return myVector;
}

Vector* newMinimalVector(Comparator comparator,
        void (*printOne)(void*)) {
    Vector* myVector = malloc(sizeof *myVector);
    myVector -> length = 0;
    myVector -> data = NULL;
    myVector -> get = get;
    myVector -> add = add;
    myVector -> remove = del;
    myVector -> print = print_vector;
    myVector -> printOne = printOne;
    myVector -> search = linear_search_vector;
    myVector -> comparator = comparator;

    return myVector;
}

Vector* _newMinimalVectorWithArgs(Comparator comparator, void
(*printOne)(void*), size_t argc, ...) {
    Vector* thisVector = newMinimalVector(comparator, printOne);

    va_list valist;

    // va_start takes the valist and the last parameter before the "..."
    va_start(valist, argc);
    for (int i = 0 ; i < argc ; i++) {
        // printf("*%d*", va_arg(valist, int));
        thisVector -> add (thisVector, va_arg(valist, void*));
    }
    va_end(valist);
    return thisVector;
}

void init(Vector *list,
        void (*print_util)(Vector*),
```

```c
        int (*compare)(const void*, const void*),
        int (*compare_r)(const void*, const void*)) {
    list -> get = get;
    list -> length  = 0;
    list -> data    = NULL;
    list -> print   = print_util;
    list -> add     = add;
    list -> remove  = del;
    list -> comparator = compare;
    list -> comparator_r = compare_r;
    list -> sort = sort;
    list -> search = linear_search_vector;
}

void* get(Vector *list, int loc) {
    return (list -> data)[loc];
}

void add(Vector *list, void * DATA) {
    list -> data = realloc(list -> data, sizeof *(list -> data) * (list ->
length + 1));
    (list -> data)[(list -> length)] = DATA;
    list -> length++;
}

void del(Vector *list, int index) {
    for (int i = index ; i < list -> length - 1 ; i++) {
        (list -> data)[i] = (list -> data)[i+1];
    }
    list -> length --;
    list -> data = realloc(list -> data, sizeof *(list -> data) * (list ->
length));
}

void sort(Vector *list, bool descending) {
    if (descending)
        qsort(list -> data, (size_t)list -> length,
            sizeof *(list -> data),
            list -> comparator_r);
    else
        qsort(list -> data, (size_t)list -> length,
            sizeof *(list -> data),
            list -> comparator);
}

/* Searching Algorithms */
int linear_search_vector(Vector* list, void* param, int (*comparator)(const
void*, const void*)) {
    for (int i = 0 ; i < list -> length ; i++) {
        if (comparator(list -> data[i], param) == 0) {
            list->printOne(list->data[i]);
            return i;
        }
    }

    return -1;
}
```

```c
/* Comparator Functions */
int int_compare(const void * a1, const void * a2) {
    // di(**((const int**)a1));
    if (**((const int**) a1) > **((const int**) a2)) return 1;
    if (**((const int**) a1) < **((const int**) a2)) return -1;
    return 0;
}

int int_compare_r(const void * a1, const void * a2) {
    // di(**((const int**)a1));
    if (**((const int**) a1) > **((const int**) a2)) return -1;
    if (**((const int**) a1) < **((const int**) a2)) return 1;
    return 0;
}

int string_compare(const void * s1, const void * s2) {
    return strcmp(*(const char**)s1, *(const char**)s2);
}

int string_compare_r(const void * s1, const void * s2) {
    return (-1)*strcmp(*(const char**)s1, *(const char**)s2);
}

/* Generic Print Utility */
void print_vector(Vector* list) {
    for (int i = 0 ; i < list -> length ; i++) {
        list -> printOne(list -> data[i]);
    }
}

/*Print Utils*/

void print_int_vector(Vector *list) {
    for (int i = 0 ; i < list -> length ; i++) {
        di(*(int *)((list -> data)[i]));
    }
}

void print_string_vector(Vector *list) {
    for (int i = 0 ; i < list -> length ; i++) {
        ds((char *)((list -> data)[i]))
    }
}

/* Single Data Print Utils */
void print_int(void* data) {
    printf("%d", (*(int*)data));
}

void print_string(void* data) {
    printf("%s", ((char*)data));
}

/* New Data of Specific DataType Methods */

void* new_int(int data) {
    int* new_data = malloc(sizeof *new_data);
    *new_data = data;
    return new_data;
}

void* new_double(double data) {
    double* new_data = malloc(sizeof *new_data);
    *new_data = data;
    return new_data;
}

void* new_string(char* data) {
    char* new_data = malloc(strlen(data) * sizeof *new_data);
    strcpy(new_data, data);
    return new_data;
}

/* Other Sorting Techniques */
void insertion_sort(void** arr, size_t length, size_t ele_size, int
(*comparator)(const void* data1, const void* data2)) {
    void* key;
    for (int i = 1 ; i < length ; i++) {
        key = arr[i];
        int j = i - 1;
        while (j >= 0 && comparator(arr[j], key) >= 0) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

vector.h

```c
#ifndef VECTOR_H
#define VECTOR_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "vector.h"

/* For vargs without a list count */
#define PP_NARG(...) \
         PP_NARG_(__VA_ARGS__,PP_RSEQ_N())
#define PP_NARG_(...) \
         PP_ARG_N(__VA_ARGS__)
#define PP_ARG_N( \
          _1, _2, _3, _4, _5, _6, _7, _8, _9,_10, \
         _11,_12,_13,_14,_15,_16,_17,_18,_19,_20, \
         _21,_22,_23,_24,_25,_26,_27,_28,_29,_30, \
         _31,_32,_33,_34,_35,_36,_37,_38,_39,_40, \
         _41,_42,_43,_44,_45,_46,_47,_48,_49,_50, \
         _51,_52,_53,_54,_55,_56,_57,_58,_59,_60, \
         _61,_62,_63,_64,_65,_66,_67,_68,_69,_70, \
         _71,_72,_73,_74,_75,_76,_77,_78,_79,_80, \
         _81,_82,_83,_84,_85,_86,_87,_88,_89,_90, \
         _91,_92,_93,_94,_95,_96,_97,_98,_99,_100, \
         _101,_102,_103,_104,_105,_106,_107,_108,_109,_110, \
         _111,_112,_113,_114,_115,_116,_117,_118,_119,_120, \
         _121,_122,_123,_124,_125,_126,_127,N,...) N
#define PP_RSEQ_N() \
         127,126,125,124,123,122,121,120, \
         119,118,117,116,115,114,113,112,111,110, \
         109,108,107,106,105,104,103,102,101,100, \
         99,98,97,96,95,94,93,92,91,90, \
         89,88,87,86,85,84,83,82,81,80, \
         79,78,77,76,75,74,73,72,71,70, \
         69,68,67,66,65,64,63,62,61,60, \
         59,58,57,56,55,54,53,52,51,50, \
         49,48,47,46,45,44,43,42,41,40, \
         39,38,37,36,35,34,33,32,31,30, \
         29,28,27,26,25,24,23,22,21,20, \
         19,18,17,16,15,14,13,12,11,10, \
         9,8,7,6,5,4,3,2,1,0

typedef int (*Comparator)(const void*, const void*);

/* The Vector structure */
struct Vector {
    int     length;
    void    **data;
    void*    (*get)(struct Vector*, int);
    void    (*print)(struct Vector*);
    void    (*add)(struct Vector*, void *);
    void    (*remove)(struct Vector*, int);
    int     (*comparator)(const void *, const void *);
    int     (*comparator_r)(const void *, const void *);
    void    (*sort)(struct Vector*, bool);
    int     (*search)(struct Vector*, void*, int (*)(const void*, const
void*));
    void    (*printOne)(void* data);
};

typedef struct Vector Vector;

#define _COMPARATOR_ int (*)(const void*, const void*)
#define _PRINTONE_ void (*)(void*)

Vector* _newMinimalVectorWithArgs(_COMPARATOR_, _PRINTONE_, size_t argc,
...);
#define newMinimalVectorWithArgs(_COMPARATOR_, _PRINTONE_, ...) \
_newMinimalVectorWithArgs(_COMPARATOR_, _PRINTONE_,PP_NARG(__VA_ARGS__),
__VA_ARGS__)

/* Generic methods */
Vector* newVector(void (*)(Vector*), _COMPARATOR_, _COMPARATOR_);
Vector* newMinimalVector(_COMPARATOR_, _PRINTONE_);
void* get(Vector *, int);
void add(Vector *, void *);
void init(Vector *, void (*)(Vector *), _COMPARATOR_, _COMPARATOR_);
void del(Vector*, int);
void sort(Vector*, bool descending);
void print_vector(Vector* list);

/* DataType specific methods */
void*   new_int(int);
void*   new_double(double);
void*   new_string(char*);

/* Print Utils */
void    print_int_vector(Vector *);
void    print_string_vector(Vector *);

void    print_int(void* data);
void    print_string(void* data);

/* Comparator methods */
int     int_compare(const void *, const void *);
int     int_compare_r(const void *, const void *);
int     string_compare(const void *, const void *);
int     string_compare_r(const void *, const void *);

/* Sorting Methods */
void insertion_sort(void** arr, size_t length, size_t ele_size,
_COMPARATOR_);

/* Searching Algorithms */
int linear_search_vector(Vector*, void* param, _COMPARATOR_);

#endif
```

main.c

```
#include <stdio.h>
#include "vector.h"
#include "stack.h"
#include "quack.h"
#include "debug_helper.h"

int main() {
    Quack* myquack;

    if ((myquack = newQuack()) == NULL) {
        return -1;
    }

    myquack -> push(myquack, new_int(1));
    myquack -> push(myquack, new_int(2));
    myquack -> push(myquack, new_int(3));

    printf("\nPUSH 1, 2, 3 :\n");
    myquack -> display(myquack, print_int);
    printf("\n");

    printf("\nPULL:\n");
    myquack -> pull(myquack);
    myquack -> display(myquack, print_int);
    printf("\n");

    printf("\nPOP:\n");
    myquack -> pop(myquack);
    myquack -> display(myquack, print_int);
    printf("\n");

    return 0;
}
```

debug_helper.h

```
#ifndef DEBUG_HELPER_H
#define DEBUG_HELPER_H

#define ds(s) printf("\nDEBUG--*"#s " : %s*\n", s);
#define dc(c) printf("\nDEBUG--%"#c " : %c$\n", c);
#define di(i) printf("\nDEBUG--#"#i " : %d#\n", i);

#endif
```