Data Structure Implementation Compilation

Faaiq Waqar

Bag ADT:

Descriptions and Definitions:

Definition: Maintains an unordered collection of data elements

Operations: Add data element, remove data element, check if bag contains element

Interface:

```
void initBag(container);
void addBag(container,value);
void containsBag(container,value);
void removeBag(container,value);
void sizeBag(container);
```

Header File:

```
#define TYPE double
#define MAX_SIZE 100

struct Bag{
        TYPE data[MAX_SIZE];
        int size;
};

void initBag(struct dyArr *da);
void freeBag(struct dyArr *da)
void addBag(struct dyArr *da, TYPE val);
void containsBag(struct dyArr *da, TYPE val);
void removeBag(struct dyArr *da, TYPE val);
void sizeBag(struct dyArr *da);
```

Function Implementation:

addBag - Add an element to the Bag

```
void addBag(struct Bag *b, TYPE val){
        assert(b != NULL); /*check init*/
        If (b->size >= MAX_SIZE)
```

```
                Return; /*check for memory space*/
        b->data[b->size] = val; /* add value at the end, because it's easy*/
        b->size++;  /*increment the size of the bag*/
}
```

removeBag - Remove an element from the Bag

```
void removeBag(struct Bag *b, TYPE val){
        assert(b != NULL); /*check init*/
        int i = b->size - 1; /*set index at end element*/
        While (i >= 0){ /*element search*/
                If (b->data[i] == val){
                        /* copy the last element */
                        b->data[i] = b->data[b->size -1];
                        b->size--; /*size reduction*/
                        return;
                }

                i--;
        }
}
```

removeBag (single) - Remove a single value from the bag

```
void removeDyArr(struct dyArr *da, TYPE val){
        int i = 0; /* initialize an index for the loop */
        while (i < da->size){ /*loop over the index of size*/
                if(EQ(val,da->data[i])){
                        _removeAtDyArr(da,i); /*remove at if found*/
                        return; /*return if only a single one*/
                }

                i++; /*increment per iteration*/
        }
}
```

removeBag (multiple) - Remove multiple values from the bag

```
void removeDyArr(struct dyArr *da, TYPE val){
        int i = 0;
        while(i < da->size){
                if (EQ(val,da->data[i])){
                        _removeAtDyArr(da, i);
                        i--;
                }
```

```
                i++;
        }
}
```

_removeAtDyArr - Remove a value at a Bag location

```
void _removeAtDyArr(struct dyArr *da, int idx){
        assert((da->size > idx) && (idx >= 0)); /*assert the size is bigger than idx, and inx is valid*/
        for (i = idx; i <= da->size - 2; i++){ /*end before*/
                da->data[i] = da->data[i+1]; /*copy from idx*/
        }
        da->size--; /*decrement size*/
}
```

_removeAtDyArr - Not ordered, single remove

```
void _removeAtDyArr(struct dyArr *da, int idx){
        int i;
        assert((da->size > idx) && (idx >= 0));
        /*put last element in place of removal*/
        da->data[idx] = da->data[da->size - 1];
        da->size--;
}
```

Dynamic Arrays:

Descriptions and Definitions:

Positives: Simple, Each element is accessible in O(1) runtime complexity

Negatives: Size is fixed upon creation, longer runtime complexity

Size: Current number of elements, managed by an internal data value

Capacity: Number of elements a dynamic array can hold before it must resize

Adding an element: Reallocate new space, copy all data values to the new space, hide details from the user.

Header File:

```
struct dyArr{
        TYPE* data;
        int size;
        int capacity;
};
```

```
void initiDynArr(struct dyArr *da, int cap);
void freeDynArr(struct dyArr *da);
void addDynArr(struct dyArr *da, TYPE val);
void removeDynArr(struct dyArr *da, TYPE val);
void _dyArrDoubleCapacity(struct dyArr *da);
…
```

Function Implementation:

initDynArr - Initialization of dynamic array

```
void initiDynArr(struct dyArr *da, int cap){
        assert(cap >= 0); /*works with only certain assertion*/
        da->capacity = cap; /*must be given a capacity*/
        da->size=0; /*nothing stored*/
        da->data = (TYPE*); /*setting data dynamic array*/

        malloc (da->capacity * sizeof(TYPE)); /*allocation*/
        assert(da->data != 0); /*status check*/
}
```

freeDynArr - Clean up dynamic array

```
void freeDynArr(struct dyArr *da){
        Assert (da != 0); /*check alloc */
        free(da->data); /*free array*/
        da->capacity = 0; /*set integer values to 0*/
        da->size = 0;
}
```

addDynArr - Add new element to array

```
void addDynArr(struct dyArr *da, TYPE val){
        /*check capacity, if not, call the double*/
        if(da->size >= da->capacity)
                _dyArrDoubleCapacity(da);
        da->data[da->size] = val;
        da->size++; /*size increasement*/
}
```

_dyArrDoubleCapacity - Double the capacity of array

```
void _dyArrDoubleCapacity(struct dyArr *da){
        TYPE *oldbuffer = da->data; /*use address for old*/
        int oldsize = da->size;

        /*allocate new memory*/
```

```
        initDynArr(da, 2 * da->capacity);
        for(int i = 0; i < oldsize; i++)
                da->data[i] = oldbuffer[i];
        da->size = oldsize;
        free(oldbuffer); /*free the old memory*/
}
```

Stacks:

Descriptions and Definitions:

Definition: Maintains a collection of data elements in last in first out format (LIFO).

Operations: Add element to the stack, remove an element from the stack, read the top element of the stack, check if an element is contained in the stack.

Interface:

```
void initStack(container);
void pushStack(container, value);
void topStack(container);
void popStack(container);
void isEmptyStack(container);
```

Header File:

```
struct dyArr{
        TYPE *data;
        int size;
        int capacity;
};

/*function prototypes*/
/*interface of DA*/
void initDynArr(struct dyArr *da, int cap);
void freeDynArr(struct dyArr *da);
void addtDynArr(struct dyArr *da, TYPE val);
TYPE getDynArr(struct dyArr *da, int idx);
TYPE putDynArr(struct dyArr *da, int idx ,TYPE val);
int sizeDynArr(struct dyArr *da);
void _dyArrDoubleCapacity(struct dyArr *da);
/*interface of Stack*/
void initStack(struct dyArr *da, int cap);
void freeStack(struct dyArr *da);
void pushStack(struct dyArr *da, TYPE d);
TYPE topStack(struct dyArr *da);
```

```
void popStack(struct dyArr *da);
int isEmptyStack(struct dyArr *da);
```

Function Implementation:

initStack - Initialization of a stack

```
void initStack(struct dyArr *da, int cap){
        initDynArr(da,cap);
}
```

freeStack - Clean up of the stack

```
freeStack(struct dyArr *da){
        freeDynArr(da);
}
```

isEmptyStack - Does a return verifying at 0

```
int isEmptyStack(struct dyArr *da){
        return (sizeDynArr(da) == 0);
}
```

pushStack - Add new element to stack

```
void pushStack(struct dyArr *da, TYPE val){
        addDynArr(da,val);
}
```

topStack - Reads the top elements of stack

```
TYPE topStack(struct dyArr *da){
        assert(sizeDynArr(da) > 0);
        /*assert stack is not empty*/
        return getDynArr(da, sizeDynArr(da)-1);
}
```

popStack - Moves pointer to the top element

```
void popStack(struct dyArr *da){
        assert(sizeDynArr(da) > 0); /*assert that it is not already empty*/
        da->size--; /*decrement size*/
};
```

Queues:

Descriptions and Definitions:

Definition: Elements are inserted at one end, and are removed from another, Operating first in first out (FIFO).

Removing Elements: Requires moving elements, with O(n)

Insertion: Insertion is done to the end had O(1) complexity

Interface:

void addBack(newElement);
TYPE front();
void removeFront();
void isEmpty():

Deques:

Descriptions and Definitions:

Definition: Insertions at both front and back, as well as removals from both the front and the back, unlike queues, Comparable to 2 end to end stacks, a combination of stack and queue.

Key Idea: Do not tie the front to index zero, alloc both "front" and "back" to float around the array.

Interface:

addFront(newElem); - Insertion to front
addBack(newElem); - Insertion to back
front(); - Returns front element
back(); - Returns back element
removeFront(); - Removal of front
removeBack(); - Removal of back

Header File:

```
struct deque {
        TYPE *data;
        int capacity;
        int size;
        int start;
}
```

Function Implementation:

initDeque - Creating a deque object

```
void initDeque (struct deque *d, int cap) {
        assert(cap > 0);
        d->size = d->start = 0; /*Initially no data in Deque*/
        d->capacity = cap;

        d->data = (TYPE *) malloc(cap * sizeof(TYPE));
```

```
        assert(d->data!=0);
}
```

addBackDeque - Inserting element to backend

```
void addBackDeque(struct deque *d, TYPE val){
        int back_idx;
        if(d->size == d->capacity)
                _doubleCapDeque(d);

        /* Increment the back index */
        back_idx = (d->start + d->size) % d->capacity; /*used to determine the back idx*/
        d->data[back_idx] = val; /*add to the back of the deque*/
        d->size++;
}
```

addFrontDeque - Inserting element to frontend

```
void addFrontDeque(struct deque *d, TYPE val){
        if(d->size == d->capacity)
                _doubleCapDeque(d);

        /* Decrement the front index */
        d->start--;
        if(d->start < 0)
                d->start += d->capacity;
        d->data[d->start] = val;
        d->size++;
}
```

Linked Lists:

Descriptions and Definitions:

Problems with Dynamic Arrays: In dynamic arrays, data is kept in a single large block of memory, and often more memory is used than necessary.

Positives of Linked List Use: Good alternative that has memory use that is always proportional to the number of elements in the collection.

Characteristic of Linked Lists: Elements are held in objects called links, likes are 1 to 1 with data elements, allocated and released as necessary.

Elements of Linked Lists: Sentinel, a special link for start or end, which points to first and/or last link, depending on single or doubly linked lists.

Header File:

```
struct Link{
        TYPE value;
        struct Link *next;
};
```

List Stack:

Descriptions and Definitions:

Description: Sentinel points to the first element, and it points to NULL if the stack is empty. Add or remove elements only from front. Add or remove elements only from the front. Allow only singly linked list. Can access only the first element.

Implementation:

```
struct Link{
        TYPE value;
        struct Link *next;
};

struct ListStack{
        struct Link *sentinel;
};
```

Function Implementation:

InitStack - Initializing a list stack

```
void InitStack(struct ListStack *stk){
        /* initialize the sentinel */
        struct Link *sentinel =
                (struct Link *)malloc(sizeof(struct Link));
        assert(sentinel != 0);

        /* linked list is empty */
        sentinel->next = NULL;
        stk->sentinel = sentinel;
}
```

pushStack - Push element onto the list stack

```
void pushStack(struct listStack *stk, TYPE val){
        struct Link *new =
                (struct Link *) malloc(sizeof(struct Link));
        assert (new != 0);
        new->value = val;
        new->next = stk->sentinel->next;
        stk->sentinel->next = new;
```

```
}
```

popStack - Pop element off the list stack

```
void popStack(struct listStack *stk){
	struct Link *lnk = stk->sentinel->next;
	if(lnk!=NULL){
		stk->sentinel->next = lnk->next;
		free(lnk);
	}
}
```

List Bag:

Descriptions and Definitions:

Description: Initialize, and add operations which are similar to a list stack, contains and remove operations are tricky, and links need to be patched up after element removal.

Header File:

```
struct Link{
	TYPE value;
	struct Link *next;
};

struct ListBag{
	struct Link *sentinel;
};
```

Function Implementation:

removeListBag [First Occurance] - Removes specified element from list bag

```
void removeListBag(struct ListBag *b, TYPE val){
	struct Link *previous = b->sentinel;
	struct Link *current = b->sentinel->next;
	while (current != NULL){
			if (EQ(current->value,val){
				previous->next = current->next;
				free(current);
				return;
		}
		previous = current;
		current = current->next;
	}
}
```

removeListBag [All Occurances] - Removes specified element from list bag

```
void removeListBag(struct ListBag *b, TYPE val){
	struct Link *previous = b->sentinel;
	struct Link *current = b->sentinel->next;
	while (current != NULL){
			if (EQ(current->value,val){
				previous->next = current->next;
				free(current);
				current = previous;
		}
		previous = current;
		current = current->next;
	}
}
```

List Queue:

Descriptions and Definitions:

Description: Add elements to the tail of the list, and remove elements from the head of the list

Header File:

```
struct Link{
	TYPE value;
	struct Link *next;
}

struct listQueue{
	struct Link *head;
	struct Link *tail;
}
```

Function Implementation:

addBackListQueue - Adds element to the back of list queue

```
void addBackListQueue(struct ListQueue *q, TYPE val){
	assert(q);
	struct Link *lnk =
		(struct Link *) malloc(sizeof(struct Link));
	assert(lnk != 0);
	lnk->next = 0;
	lnk->value = val;
	q->tail->next = lnk;
	q->tail -> lnk;
}
```

removeListQueue - Remove element from the list queue

```
removeListQueue(struct ListQueue *q){
        struct link * lnk = q->head->next;
        if( ! isEmptyListQueue(q)){
                q->head->next = lnk->next;
                if(q->head->next == 0)
                        q->tail = q->head;
                free(lnk);
        }
}
```

Doubly Linked Lists:

Header File:

```
struct dlink{
        TYPE value;
        struct dlink *next;
        struct dlink *previous;
};
```

Doubly Linked Deques:

Descriptions and Definitions:

Description: Add and remove from both the front and back of the deque, and must maintain forward and backward links

If One Sentinel is Good: Add a sentinel to both front and back, which eliminates the need to handle special cases.

Interface:

```
int isEmpty();
void addFront(TYPE val);
void addBack(TYPE val);
void removeFront();
void removeBack();
TYPE front();
TYPE back();
```

Header File:

```
struct dlink{
        TYPE value;
        struct dlink *next;
        struct dlink *prev;
```

```
};

struct listDeque{
        int size;
        struct dlink * frontSentinel;
        struct dlink * backSentinel;
};
```

Function Implementation:

_addDeque - Add a double link to Deque

```
_addDeque(struct listDeque *dq, struct dlink *lnk, TYPE e){
        /* Allocate Memory */
        assert(dq);
        struct dlink *newlink =
                (struct dlink *) malloc(sizeof(struct dlink));
        assert(newlink != 0);
        newlink->value = e;

        /* Connect to New Link */
        newlink->prev = lnk->prev;
        newlink->next = lnk;
        lnk->prev->next = newlink;
        lnk->prev = newlink;
        dq->size++;
}
```

addFrontDeque - Add element to front of the list deque

```
void addFrontDeque(struct listDeque *dq, TYPE e){
        _addDeque(dq, dq->frontSentinel->next, e);
}
```

addBackDeque - Add element to back of the list deque

```
void addBackDeque(struct listDeque *dq, TYPE e){
        _addDeque(dq, dq->backSentinel, e);
}
```

Doubly List Bags:

Interface:

```
void initBag();
int isEmptyBag();
void addBag();
void removeBag();
```

```
void containsBag();
```

Header File:

```
struct dlink{
        TYPE value;
        struct dlink *next;
        struct dlink *prev;
};

struct listDBag{
        int size;
        struct dlink *frontSentinel;
        struct dlink *backsentinel;
};
```

Function Implementation:

initDBag - Initialize doubly list bag

```
void initDBag(struct listDBag *db){
        assert(db);
        db->frontSentinel =
                (struct dlink *) malloc(sizeof(struct dlink));
        assert(db->frontSentinel != 0);

        db->backSentinel =
                (struct dlink *) malloc(sizeof(struct dlink));
        assert(db->backSentinel != 0);

        db->frontSentinel->next = db->backSentinel;
        db-> backSentinel->prev = db->frontSentinel;
        db->size = 0;
}
```

isEmptyDBag - Checks if the doubly link bag is empty

```
void isEmptyDBag(struct listDBag *db){
        assert(db);
        return db->size = 0;
}
```

addDBag - Add an element to the doubly link bag

```
void addDBag(struct listDBag *db, TYPE e){
        assert(db);
        _addDList(db, db->frontSentinel->next, e);
}
```

containsDBag - Checks doubly link bag for specified element

```
int containsDBag(struct listDBag *db, TYPE e){
	struct dlink *current;
	assert(!isEmptyBag(db));
	current = db->frontSentinel->next;
	while(current != db->backSentinel){
		if(current->value == e)
			return 1;
		current = current->next;
	}
	return 0;
}
```

Binary Search Algorithm:

Function Implementation:

```
int binarySearch(TYPE *data, int size, TYPE val){
	int low = 0;
	int high = size;
	int mid = 0;

	while(low < high){
		mid = (low + high)  / 2;
		if (data[mid] < val)
			low = mid + 1;
		else
			high = mid;
	}

	return low;
}
```

Bag Search Algorithms:

Function Implementation:

sortedContains - Search algorithm with lower runtime complexity

```
int sortedContains(struct dynArr *da, TYPE val){
	int idx = binarySearch(da->data, da->size, val);
	if(idx < da->size && da->data[idx] == val)
		return 1;
	return 0;
}
```

sortedAdd - Add to a sorted dynamic array

```
int sortedAdd(struct dynArr *da, TYPE val){
        int idx = binarySearch(da->data, da->size, val);
        _addAt(da, idx, val);
}
```

sortedRemove - Remove from a sorted dynamic array

```
int sortedRemove(struct dynArr *da, TYPE val){
        int idx = binarySearch(da->data, da->size, val);
        if(idx < da->size && da->data[idx] == val)
                _removeAt(da, idx);
}
```

Set Operation Algorithms:

Algorithm Implementation:

Intersection - Special case of merge

```
while (i < d_size && j < e_size){
        if (d[i] < e[j])
                i++;
        else if (d[i] > e[j])
                j++;
        else
                if(z[k] != d[i]){
                        /* add d[i] to z */
                        k++;
                }
        i++;
        j++;
}
```

Union - Special case of merge

```
while (i < d->size && j < e->size){
        if (d[i] < e[j]){
                if(d[i] != u[k]){
                        /*add d[i] to u; k++ */
                }
                i++;
        }
        else if (d[i] > e[j]){
                if(e[j] != u[k]){
                        /*add e[j] to u; k++ */
                }
```

```
                j++;
        }
        else{
                if(e[j] != u[k]){
                        /*add e[j] to u; k++ */
                }
                i++;
                j++;
        }
}
if (i == d->size)
        /* add rest of e to the union */
if (i == e->size)
        /* add rest of d to the union */
```

Difference - Derived from Union and Intersection

```
while (i < d->size && j < e->size){
        if (d[i] < e[j] ){
                /* add d[i] to diff */
                i++;
        }
        else if(d[i] > e[j])
                j++:
        else{
                i++:
                j++;
        }
}

if (j == e->size && i < d->size){
        /* add rest of d to diff  */
}
```

Sorted Linked Lists:

Structure Definition:

```
struct list {
        struct link *Sentinel;
        int size;
};
```

Function Implementation:

slideRightSortedList - Find an element in a sorted list

```
struct link * slideRightSortedList(struct link *current, TYPE e){
        assert(current);
        while((current->next != 0) && LESS_THAN(current->next->value, e))
                current = current->next;
        return current;
}
```

addSortedList - Add an element to a sorted linked list

```
void addSortedList (struct list *lst, TYPE e){
        assert(lst);
        struct link *current = slideRightSortedList(lst->Sentinel, e);
        struct link *newList = (struct link *) malloc(sizeof(struct link));
        assert (newLink != 0);
        newLink->value = e;
        newLink->next = current->next;
        /* For Doubly Linked Lists */
        /* newLink->previous = current; */
        /* current->next->previous = newLink; */
        current->next = newLink;
        lst->size++;
}
```

removeSortedList - Remove an element from a sorted linked list

```
void removeSortedList (struct list *lst, TYPE e){
        struct link *temp;
        assert(lst);
        struct link *current = slideRightSortedList (lst->Sentinel, e);
        if ((current->next != 0) && (EQ(current->next->value, e)) {
                temp = current->next;
                current->next = current->next->next;
                /* Doubly Linked List … */
                /* current->next->previous = current; */
                free(temp);
                lst->size--;
        }
}
```

Open Address Hash Tables:

Header File:

```
struct openHashTable {
```

```
        TYPE ** table; /* array of pointers to TYPE */
        int tableSize;
        int count;
};
```

Function Implementation:

initOpenHashTable - Initialize an open address hash table

```
void initOpenHashTable (struct openHashTable * ht, int size) {
        int i = 0;
        assert (size > 0);
        ht->table = (TYPE **) malloc(sizeof(TYPE *) * size));
        assert (ht->table != NULL);
        for (i = 0; i < size; i++)
                ht->table[i] = NULL;
        ht->tablesize = size;
        ht->count = 0;
}
```

openHashTableSize - Returns the size of an open address hash table

```
int openHashTableSize (struct openHashTable * ht, TYPE newValue) {
        return ht->count;
}
```

openHashTableAdd - Add an element to the open address hash table

```
void openHashTableAdd (struct openHashTable * ht, TYPE newValue){
        int idx = 0;
        if ((ht->count / (double) ht->tableSize) > 0.75)
                resizeOpenHashTable (ht);
        ht->count++;
        idx = HASH(newValue) % ht->tableSize;
        if (idx < 0)
                idx += ht->tableSize;
        while (ht->table[idx] != NULL){
                idx++;
                if (idx >= ht->tableSize)
                        idx = 0;
        }
        ht->table[idx] = val;
}
```

openHashTableBagContains - Checks if an element exists in open hash table

```
void openHashTableBagContains (struct openHashTable * ht, TYPE testValue){
        int idx = HASH(newValue) % ht->tableSize;
```

```
        if (idx < 0)
                idx += ht->tableSize;
        while (ht->table[idx] != 0) {
                if (EQ(testValue, *(ht->table[idx]))
                        return 1;
                idx++;
                if (idx == ht->tableSize)
                        idx = 0;
        }
        return 0;
}
```

_resizeOpenHashTable - Double the capacity of the hash table

```
void _resizeOpenHashTable (struct openHashTable * ht) {
        int oldTableSize = ht->tableSize;
        ht->tableSize = 2 * ht->tableSize;
        TYPE ** oldArray = ht->table;
        int i = 0;

        /* create a new array and populate */
        ht->table = (TYPE **) malloc(sizeof(TYPE *) * ht->tableSize));

        /* copy elements into new hash table */
        for (i = 0; i < oldTableSize; i++){
                if (oldArray[i] != NULL)
                        addOpenHashTable (ht, oldArray[i]);
        }
        free (oldArray);
}
```

Hash Tables:

Descriptions and Definitions:

Assumptions: The time to compute a hash function is constant. In a best case, all the buckets will contain the same number of element.

Complexity: The load factor of a hash table is $O(\lambda)$, where $\lambda = n/m$, where n is the number of elements, and m is the size of the table. We want to keep the load factor small, and this, if the load factor crosses a certain threshold, we should double the size of the hash table array.

When to Use: Need a guarantee that elements are uniformly distributed, otherwise, a skip list or an AVL tree might be a faster option for the data.

Interface:

```
void initHashTable();
void add HashTable();
ValueType* contains Hash Table();
void removeHashTable();
```

Header File:

```
struct HashTable{
        struct Link ** table;  /*Array of Lists*/
        int count; /*number of elements in the table*/
        int tableSize; /*the number of lists*/
};
struct Link{
        struct DataElem elem;
        struct Link * next;
};
struct DataElem{
        TYPE_KEY key;
        TYPE_VALUE value;
};
```

Function Implementation:

initHashTable - Initializing a hash table in C

```
void initHashTable (struct HashTable * ht, int size){
        /* Step One - Allocate memory into array, each representing a head */
        int index;
        ht->table = (struct Link **) malloc(sizeof(struct Link *) * size);
        assert(ht->table != 0);
        /* Step Two - Initialize headers of the linked lists */
        ht->tableSize = size;
        ht->count = 0;
        for (index = 0; index < tableSize; index++){
                ht->table[index] = NULL;
        }
}
```

addHashTable - Add an element to the hash table in C

```
void addHashTable (struct HashTable * ht, struct DataElem elem){
        /* Step One: Compute the index of the element */
        int hash = HASH(elem.key);
        int hashIndex = (int) (labs(hash) % ht->tableSize); /*labs to get long absolute integer */
        float loadFactor = ht->count / ht->tableSize;
```

```
        /* Step Two: Allocate memory to elem */
        struct Link * newLink = (struct Link *) malloc(sizeof(struct Link));
        assert(newLink);
        newLink->elem = elem;
        /*Step Three: Insert element - add to bucket at front */
        newLink->next = ht->table[hashIndex];
        ht->table[hashIndex] = newLink;
        /* Step Four: Increment the number of elements */
        ht->count++;
        if (loadFactor > MAX_LOAD_FACTOR)
                _resizeTable(ht);
}
```

_resizeTable - Double the size of the table in cases of too large load factor

```
void _resizeTable(struct HashTable * ht){
        int oldSize = ht->tableSize;
        struct HashTable * oldht = ht;
        struct Link *cur;
        struct Link *last;
        int i = 0;

        /* Create new memory location */
        initHashTable(ht, 2*oldSize);
        /* Copy old table */
        for ( i = 0; i < oldSize; i++){
                cur = oldht->table[i];
                while(cur != NULL){
                        addHashTable(ht, cur->elem);
                        last = cur;
                        cur = cur->next;
                        free(last);
                }
        }
        free(oldht);
}
```

containsHashTable - Checks hash table for a specific element

```
int containsHashTable (struct HashTable * ht, struct DataElem elem){
        int hash = HASH(elem.key);
        int hashIndex = (int) (labs(hash) % ht->tableSize);
        struct Link * cur;

        cur = ht->table[hashIndex]; /*Goes to the indexed bucket */
```

```
        while(cur != NULL){
                if(EQ(cur->elem.value, elem.value))
                        return 1;
                cur =  cur->next;
        }
        return 0;
}
```

removeHashTable - Removes a specified data element in the hash table

```
void removeHashTable (struct HashTable * ht, struct DataElem elem){
        int hash = HASH(elem.key);
        int hashIndex = (int) (labs(hash) % ht->tableSize);
        struct Link * cur = ht->table[hashIndex];
        struct Link * last = ht->table[hashIndex];

        while(cur != NULL){
                if(EQ(cur->elem.value, elem.value)){
                        if(cur == ht->table[hashIndex])
                                ht->table[hashIndex] = cur->next;
                        else
                                last->next = cur->next;
                        free(cur);
                        cur = NULL;
                        ht->count--;
                }
                else{
                        last = cur;
                        cur = cur->next;
                }
        }
}
```

stringHash1 - Example hashing function for a string

```
int stringHash1 (char * str){
        int i = 0;
        int r = 0;
        for (i = 0; str[i] != '\0'; i++)
                r += str[i];
        return r;
}
```

stringHash2 - Second example hashing function for a string

```
int stringHash2 (char * str){
```

```
        int i = 0;
        int r = 0;
        for (i = 0; str[i] != '\0'; i++)
                r +=  (i+1) * str[i];
        return r;

}
```

Binary Search Trees:

Descriptions and Definitions:

Functionality of BST: Every node value is greater than all of its left descendants, less than or equal to all of its right descendants.

Complexity: If a binary search tree is reasonably full, searching for an element will have time O(log n).

Removal Fact: When filling a hole created by an element removal, the hole is filled by the leftmost descendant of the right child.

Header File:

```
struct Node {
        TYPE val;
        struct Node * left;
        struct Node * right;
};

struct BST {
        struct Node * root;
        int size;
};
```

Interface:

```
void initBST(struct BST * tree);
int contains BST(struct BST * tree, TYPE val);
void addBST(struct BST * tree, TYPE val);
void removeBST(struct BST * tree, TYPE val);
int size BST(struct BST * tree);
```

Function Implementation:

containsBST - Checks the binary search tree for a specific value

```
int containsBST (struct BST * tree, TYPE val){
        struct Node * current = tree->root; /* sets  a pointer to the root of the tree */
        while (current != NULL) { /* loops until it has traverse tree materials */
```

```
                if (EQ(val, current->value)) /* if the value you seek is found */
                        return 1; /*return a true */
                if (LT(val, current->value)) /* if the value is less than the value in the current node */
                        current = cur->left; /*set to the left node */
                else
                        current = current->right; /* otherwise set to the right node */
        }
        return 0;
}
```

containsBST & _containsNode - Checks the binary search tree for a specific value recursively

```
int containsBST (struct BST * tree, TYPE val){
        assert (tree); /* assert tree allocation */
        if (tree->root) /* check that a root exists */
                return _containsNode(tree->root, val); /* use recursive function with node */
        else
                return 0;  /* if the tree is empty */
}

int _containsNode (struct Node * current, TYPE val) {
        int flag = 0; /* create a flag with original null value */
        if (EQ(current->value, val)) /* if the value is here, set the flag true */
                flag = 1;
        else if (LT(current->value, val) && current->left) /*if less, send flag equal to the rec left */
                flag = _containsNode(current->left, val);
        else if (current->right)
                flag = _containsNode(current->right, val); /*if right exists, send right */
        return flag; /* return the flag */
}
```

addBST - Calls a utility routine to add a node to binary search tree

```
void addBST(struct BST * tree, TYPE val) {
        assert (tree); /* assert allocation */
        tree->root = _addNode (tree->root, val); /* call recursive function on the root */
        tree->size++; /* increment the size */
}
```

_addNode - Auxiliary function used to add single node to binary search tree

```
struct Node * _addNode (struct Node * current, TYPE val) {
        if (current == NULL){ /* implement actual adding if empty space found */
                struct Node * new = (struct Node *) malloc(sizeof(struct Node)); /* alloc */
                assert(new != NULL); /* assert alloc */
                new->value = val; /* set value to parameter */
```

```
                new->left = new->right = NULL; /*set its children to NULL */
        return new; /* return the new node */
        }
        else{
                /* call recursion left or right */
                if (val < current->val) /* call recursion based on the value parameter */
                        current->left = _addNode(current->left, val);
                else
                        current->right = _addNode(current->right, val);
        }
        return current;
}
```

removeBST - Calls a utility routine to remove a node from binary search tree

```
void removeBST (struct BST * tree, TYPE e){
        if (containsBST (tree, e)) { /* Check if the value exists in the tree */
                tree->root = _removeNode(tree->root, e); /* Call recursive function using root */
                tree->size--; /* decrement size */
        }
}
```

_removeNode - Auxiliary function used to remove single node from binary search tree

```
struct Node * _removeNode (struct Bode * current, TYPE e) {
        struct Node * node; /* Create a pointer to the address of node */
        assert (current); /* assert allocation */

        if (LT(e, current->val) == 0) { /* ask here */
                if (current->right == 0){
                        node = current->left;
                        free (current);
                        return node;
                }
                current->val = _leftMost (current->right);
                current->right = _removeLeftMost (current->right);
        }
        else if  (LT(e, current->val) < 0)
                current->left = _removeNode (current->left, val);
        else
                current->right = _removeNode (current->right, val);
        return current;
}
```

_leftMost - Returns the value of the leftmost child of the current node

```
TYPE _leftMost (struct Node * current){
        while (current->left != NULL) /* while there are no more left to traverse*/
                current = current->left; /* increment */
        return current->val; /* return the value */
}
```

_removeLeftmost - Removes the leftmost descendent of current

```
struct Node * _removeLeftmost (struct Node * current) {
        struct Node * temp;
        if (current->left != NULL){
                current->left = _removeLeftmost (current->left);
                return current;
        }
        temp = current->right;
        free (current);
        return temp;
}
```

AVL Trees:

Descriptions and Definitions:

Definition: The Adelson-Velskii and Landis Tree is designed to maintain the height balanced property of Binary Search Trees. When unbalanced, we perform a "rotation" to balance the tree.

Sorting: An AVL tree can sort a collection of values. First, the data is copied into the AVL tree with complexity O(nlogn). Then, they are copied out using the in-order traversal method, that had complexity O(n). The total execution time is O(nlogn), watches quicksort benchmarks, and is less erroneous as a whole. However, it requires extra storage to maintain the tree structure.

Header File:

```
struct AVLNode {
        TYPE val;
        struct AVLNode * left;
        struct AVLNode * rightl
        int height;
};

struct AVLTree {
        struct AVLNode * root;
        int count;

};
```

Function Implementation:

_height - Gets the height of the specified node path

```
int _height (struct AVLNode * current) {
        if (current == NULL)
                return -1;
        return current->height;
}
```

_setHeight - Compute the height of specified node path

```
void _setHeight (struct AVLNode * current) {
        int lh = _height (current->left);
        int rh = _height (current->right);
        if (lh <  rh)
                current->height = 1 + rh;
        else
                current->height = 1 + lh;

}
```

_rotateLeft - Make a leftwards rotation of the AVL tree

```
struct AVLNode * _rotateLeft (struct AVLNode * current) {
        struct AVLNode * newtop = current->right;
        current->right = newtop->left;
        newtop->left = current;
        _setHeight (current);
        _setHeight (newtop);
        return newtop;
}
```

_rotateRight - Make a rightwards rotation of the AVL tree

```
struct AVLNode * _rotateRight (struct AVLNode * current) {
        struct AVLNode * newtop = current->left;
        current->left = newtop->right;
        newtop->right = current;
        _setHeight (current);
        _setHeight (newtop);
        return newtop;
}
```

balance - Balance an unbalanced node in AVL tree

```
struct AVLNode * balance (struct AVLNode * current) {
```

```
        int rotation = _height (current->right) - _height (current->left);
        int dlrotation = _height (current->left->right) - _height (current->left->left);
        int drrotation = _height (current->right->right) - _height (current->right->left);
        if (rotation < -1){
                /* (double) rotation right */
                if (dlrotation > 0){
                        /* left child is heavy on the right */
                        current->left = _rotateLeft (current->left);
                }
                return _rotateRight (current);
        }
        else if (rotation > 1){
                /* (double) rotation left */
                if (drrotation < 0) {
                        current->right = _rotateRight (current->right);
                }
                return _rotateLeft (current);
        }
        _setHeight (current);
        return current;
}
```

_addAVLNode - Add a node to the AVL tree using recursive function

```
struct AVLNode * _addAVLNode (struct AVLNode * current, TYPE e) {
        struct AVLNode * newnode;
        if (current == NULL) {
                newnode = (struct AVLNode *) malloc(sizeof(struct AVLNode));
                assert (newnode != NULL);
                newnode->value = e;
                newnode->left = newnode->right = NULL;
                return newnode;
                 /* allocate memory for newnode, add newnode to tree */

        }
        else {
                if (LT(e, current->value))
                        current->left = _addAVLNode (current->left, e);
                else
                        current->right = _addAVLNode (current->right, e);
        }
        return balance (current);
}
```

removeAVLTree - Utility function for removal of single node in an AVL tree

```
void removeAVLTree (struct AVLTree * tree, TYPE val) {
        if (containsAVLTree (tree, val)) {
                tree->root = _removeNode (tree->root, val);
                tree->count--;
        }
}
```

_removeNode - Auxiliary function used for the removal of node in AVL

```
struct AVLNode * _removeNode (struct AVL * current, TYPE e){
        struct AVLNode * temp;
        assert (current);

        if (EQ (e, current->val)){
                /* replace current with the leftmost descendant of the right child */
                if (current->right != 0){
                        current->val = _leftMost (current->right);
                        current->right = _removeLeftmost (current->right);
                        return _balance(current);
                }
                else{
                        temp = current->left;
                        free (current);
                        return temp;
                }
        }
        else if (LT(e, current->val))
                current->left = _removeNode(current->left, e);
        else
                current->right = _removeNode(current->right, e);

        return balance(current);
}
```

Priority Queue ADT:

Interface:

```
void add(newValue);
TYPE getFirst();
void removeFirst();
```

Heap ADT:

Header File:

```
struct DynArr {
	TYPE * data;
	int size;
	int capacity;
};

struct DynArr * heap;
```

Function Implementation:

addHeap - Add an element to the Heap ADT

```
void addHeap (struct DynArr * heap, TYPE val) {
	int pos = heap->size;
	int pidx = (pos - 1) / 2;
	TYPE parentVal;
	assert (heap);

	if (pos >= heap->capacity)
		_doubleCapacity (heap, 2 * heap->capacity);
	if (pos > 0)
		parentVal = heap->data[pidx];
	while (pos > 0 && LT(val, parentVal)) {
		heap->data[pos] = parentVal;
		pos = pidx;
		pidx = (pos - 1) / 2;
		if (pos > 0)
			parentVal = heap->data[pidx];
	}

	heap->data[pos] = val;
	heap->size++;
}
```

removeMinHeap - Remove the smallest element in the heap

```
void removeMinHeap (struct DynArr * heap){
	int last = heap->size-1;
	assert (heap);
	if (last != 0)
		heap->data[0] = heap->data[last];
	heap->size--;
	/* rebuild heap property */
```

```
        _adjustHeap (heap, last-1, 0); /* recursion */
}
```

_adjustHeap - Adjusts the heap during management of values

```
void _adjustHeap (struct DynArr * heap, int maxIdx, int pos) {
        int leftIdx = pos * 2 + 1;
        int rghtIdx = pos * 2 + 2;
        int smallIdx;
        assert (heap);

        if (rghtIdx < maxIdx) {
                smallIdx = _minIdx(heap, leftIdx, rightIdx);
                if (LT(heap->data[smallIdx], heap->data[pos])){
                        _swap (heap, pos, smallIdx);
                        _adjustHeap (heap, maxIdx, smallIdx);
                }
        }
        else if (leftIdx < maxIdx) {
                if(LT(heap->data[leftIdx],heap->data[pos])){
                        _swap (heap, pos, leftIdx);
                        _adjustHeap (heap, maxIdx, leftIdx);

                }
        }
}
```

_swap - Swap elements within a heap

```
void _swap (struct DynArr * da, int i, int j) {
        TYPE temp = da->data[i];
        da->data[i] = da->data[j];
        da->data[j] = temp;
}
```

_minIdx - Returns the minimum index within the heap

```
int _ minIdx (struct DynArr * da, int i, int j) {
        if (LT(da->data[i], da->data[j]))
                return i;
        return j;
}
```

buildHeap - Construct a heap out of a standard dynamic array

```
void buildHeap(struct dynArray * da){
        int maxIdx = da->size;
```

```
	int i;
	for (i = maxIdx / 2 -1; i >= 0; i--)
		_adjustHeap(da, i, maxIdx);
}
```

heapSort - Sort a heap dynamic array by descending order

```
void heapSort (struct dynArray * data) {
	int i;
	buildHeap(data);
	for (i = data->size-1; i > 0; i--){
		swapDynArr(data,0,i);
		_adjustHeap(data,0,i);
	}
}
```

addHeap - Auxiliary call to recursive add function

```
void addHeap (struct DynArr * heap, TYPE e) {
	assert (heap);
	if (heap->size >= heap->cap)
		_doubleCapacity (heap , 2 * heap->cap);
	_heapAdd(heap, e, heap->size);
	heap->size++;
}
```

_heapAdd - Recursive function to add an element to the heap

```
void _heapAdd (struct DynArr * heap, TYPE e, int index) {
	int pidx = (index - 1) / 2;
	if (index > 0 && LT(e, heap->data[pidx])){
		heap->data[index] = heap->data[pidx];
		_heapAdd (heap, e, pidx);
	}
	else {
		heap->data[index] = e;
	}
}
```

rmHeap - Remove members of a heap

```
void rmHeap (struct DynArr * heap, struct DynArr * da)  {
	int i, j;
	for (i =0; i < da->size; i++) {
		j = 0;
		while (j < heap->size) {
			if (da->data[i] == heap->data[j]){
```

```
                                _swap (heap, j, heap->size -1);
                                _adjustHeap (heap, j , heap->size);
                                heap -> size--;
                        }
                }
                j++;
        }
}
```

Graph ADT:

Function Implementation:

warshall - Implementation of Warshall's reachability algorithm:

```
void warshall (int a[N] [N]) {
        int i, j, k;
        for (k = 0; k < N; k++) {
                for (i = 0; i < N; i++)
                        if (a[i][k] != 0)
                                for (j = 0; j < N; j++)
                                        a[i][j] |= a[i][k] & a[k][j];
        }
}
```

dijkstra - Implementation of Dijkstra's shortest path algorithm

```
int * dijkstra (TYPE ** Graph, int size, int source) {
        TYPE distance[size];
        int visited[size];
        int previous[size];
        int min;

        for (int v = 0; v < size; v++){
                visited[v] = 0;
                previous[v] = source;

                distance[v] = Graph[source][v] ?
                                Graph[source][v] : INFINITY;
        }

        while(!allVisited(visited, size)) {
                min = findMin (visited, distance, size );
                visited[min] = 1;

                for(v = 0; v < size; v++){
```

```
                        if (!visited[v] && Graph [min] [v]) {
                                TYPE update = distance[min] + Graph[min][v];
                                if (update < distance[v]){
                                        distance[v] = update;
                                        previous[v] = min;
                                }
                        }
                }
        }
        return previous;
}
```