Queue

```
typedef struct _node {
	int val;
	struct _node* next;
} Node;

typedef struct _queue {
	Node* head;
	Node* rear;
} Queue;

Queue* initQueue() {
	Queue* q = malloc(sizeof(Queue));
	q->head = malloc(sizeof(Node));
	q->rear = q->head;
	q->head->next = NULL;
	return q;
}

void freeQueue(Queue* q) {
	Node* n = q->head;
	while (n) {
		Node* temp = n->next;
		free(n);
		n = temp;
	}
	free(q);
}

int front(Queue* q) {
	if (q->head == q->rear) {
		puts("The queue is empty, there is no front.");
		return -1;
	}

	return q->head->next->val;
}

void push(Queue* q, int val) {
	Node* n = malloc(sizeof(Node));
	n->val = val;
	n->next = NULL;
	q->rear->next = n;
	q->rear = n;
}

void pop(Queue* q) {
	Node* n = q->head->next;
	if (!n) {
		puts("The queue is empty.");
		return;
	}

	if (n == q->rear)
		q->rear = q->head;

	q->head->next = n->next;
	free(n);
}
```

Stack

```
typedef struct _node {
	int val;
```

```
	struct _node* next;
} Node;

typedef struct _stack {
	Node* head;
} Stack;

Stack* initStack() {
	Stack* stk = malloc(sizeof(Stack));
	stk->head = malloc(sizeof(Node));
	stk->head->next = NULL;
	return stk;
}

void freeStack(Stack* stk) {
	Node* n = stk->head;
	while (n) {
		Node* temp = n->next;
		free(n);
		n = temp;
	}
	free(stk);
}

int top(Stack* stk) {
	if (!stk->head->next) {
		puts("The stack is empty, not top.");
		return -1;
	}

	return stk->head->next->val;
}

void push(Stack* stk, int val) {
	Node* n = malloc(sizeof(Node));
	n->val = val;
	n->next = stk->head->next;
	stk->head->next = n;
}

void pop(Stack* stk) {
	Node* n = stk->head->next;
	if (!n) {
		puts("The stack is empty, cannot pop.");
		return;
	}

	stk->head->next = n->next;
	free(n);
}
```

AVL Tree

```
typedef struct _node {
	int val;
	int height;
	struct _node* left;
	struct _node* right;
} Node;

int height(Node* n) {
	if (!n) return 0;
	return n->height;
}

int max(int x, int y) {
	return x > y ? x : y;
}

void updateHeight(Node* root) {
	root->height = 1 + max(height(root->left), height(root->right));
}

Node* leftRotate(Node* k2) {
	Node* k1 = k2->right;
	k2->right = k1->left;
	k1->left = k2;
	updateHeight(k2);
	updateHeight(k1);
	return k1;
}

Node* rightRotate(Node* k2) {
	Node* k1 = k2->left;
	k2->left = k1->right;
	k1->right = k2;
	updateHeight(k2);
	updateHeight(k1);
	return k1;
}

Node* initNode(int val) {
	Node* n = malloc(sizeof(Node));
	n->left = n->right = NULL;
	n->height = 1;
	n->val = val;
	return n;
}

Node* balance(Node* root) {
	if (!root)
		return root;

	int balance = height(root->left) - height(root->right);
	if (balance > 1) {
		if (height(root->left->left) >= height(root->left->right)) {
			root = rightRotate(root);
		} else {
			root->left = leftRotate(root->left);
			root = rightRotate(root);
		}
```

```
	} else if (balance < -1) {
		if (height(root->right->right) >= height(root->right->left)) {
			root = leftRotate(root);
		} else {
			root->right = rightRotate(root->right);
			root = leftRotate(root);
		}
	}

	return root;
}

Node* insert(Node* root, int val) {
	if (!root)
		return initNode(val);

	if (root->val > val) {
		root->left = insert(root->left, val);
	} else {
		root->right = insert(root->right, val);
	}

	updateHeight(root);
	return balance(root);
}

int getMin(Node* root) {
	if (!root)
		return -1;

	if (!root->left)
		return root->val;

	return getMin(root->left);
}

Node* delete(Node* root, int val) {
	if (!root)
		return root;

	if (root->val > val) {
		root->left = delete(root->left, val);
	} else if (root->val < val) {
		root->right = delete(root->right, val);
	} else {
		if (root->right && root->left) {
			root->val = getMin(root->right);
			root->right = delete(root->right, root->val);
		} else {
			Node* temp = root;
			root = root->left ? root->left : root->right;
			free(temp);
		}
	}

	return balance(root);
}
```

Heap

```
typedef struct _heap {
	int currSize;
	int maxSize;
	int* arr;
} Heap;

Heap* initHeap(int maxSize) {
	Heap* h = malloc(sizeof(Heap));
	h->currSize = 0;
	h->maxSize = maxSize;
	h->arr = malloc(sizeof(int) * (1 + maxSize));
	return h;
}

void freeHeap(Heap* h) {
	free(h->arr);
	free(h);
}

void percolateDown(Heap* h, int hole) {
	int val = h->arr[hole];

	int child;
	for (; hole * 2 <= h->currSize; hole = child) {
		child = hole * 2;
		if (child != h->currSize && h->arr[child + 1] < h->arr[child])
			child += 1;
		if (h->arr[child] < val)
			h->arr[hole] = h->arr[child];
		else break;
	}

	h->arr[hole] = val;
}

Heap* buildHeap(int* arr, int arrLen, int heapLen) {
	Heap* h = malloc(sizeof(Heap));
	h->maxSize = heapLen;
	h->currSize = arrLen;
	h->arr = malloc(sizeof(int) * (heapLen + 1));

	int i;
	for (i = 0; i < arrLen; i++)
		h->arr[i + 1] = arr[i];

	for (i = h->currSize / 2; i > 0; i--)
		percolateDown(h, i);

	return h;
}

int findMin(Heap* h) {
	if (h->currSize == 0) {
		puts("The heap is empty, cannot findMin.");
		return -1;
	}

	return h->arr[1];
}
```

```
void insert(Heap* h, int val) {
	if (h->currSize == h->maxSize) {
		puts("The heap is full, cannot insert.");
		return;
	}

	h->currSize++;

	int hole = h->currSize;
	for (; hole > 1 && h->arr[hole / 2] > val; hole /= 2)
		h->arr[hole] = h->arr[hole / 2];
	h->arr[hole] = val;
}

void deleteMin(Heap* h) {
	if (h->currSize == 0) {
		puts("The heap is empty, cannot deleteMin.");
		return;
	}

	h->arr[1] = h->arr[h->currSize];

	h->currSize--;

	percolateDown(h, 1);
}
```

Hash Set

```
typedef struct _entry {
	enum Type type;
	int key;
} Entry;

typedef struct _hash_set {
	int currSize;
	int maxSize;
	Entry* arr;
} HashSet;

int isPrime(int num) {
	int i;
	for (i = 2; i * i <= num; i++) {
		if (num % i == 0)
			return 0;
	}
	return 1;
}

int nextPrime(int num) {
	while (!isPrime(num)) {
		num++;
	}
	return num;
}

Entry* initEntryList(int size) {
	Entry* entry = malloc(sizeof(Entry) * size);
	int i;
	for (i = 0; i < size; i++)
		entry[i].type = EMPTY;
	return entry;
}

HashSet* initHashSet(int size) {
	HashSet* hs = malloc(sizeof(HashSet));
	size = nextPrime(size);
	hs->arr = initEntryList(size);
	hs->currSize = 0;
	hs->maxSize = size;
	return hs;
}

void freeHashSet(HashSet* hs) {
	free(hs->arr);
	free(hs);
}

int hash(HashSet* hs, int key, int probes) {
	return ((key % hs->maxSize) + probes * probes) % hs->maxSize;
}

int find(HashSet* hs, int key) {
	int probes = 0;
	int hashVal = hash(hs, key, probes);
	while (hs->arr[hashVal].type == DELETED ||
		(hs->arr[hashVal].type == ACTIVE && hs->arr[hashVal].key != key)) {
		probes++;
```

```
            hashVal = hash(hs, key, probes);
      }
      if (hs->arr[hashVal].type == EMPTY) {
            return 0;
      }
      return 1;
}

HashSet* insert(HashSet* hs, int key) {
      int probes = 0;
      int hashVal = hash(hs, key, probes);
      while (hs->arr[hashVal].type == ACTIVE) {
            probes++;
            hashVal = hash(hs, key, probes);
      }

      hs->arr[hashVal].type = ACTIVE;
      hs->arr[hashVal].key = key;

      hs->currSize++;

      if (hs->currSize * 2 >= hs->maxSize) {
            HashSet* old = hs;
            hs = initHashSet(nextPrime(old->maxSize * 2));
            int i;
            for (i = 0; i < old->maxSize; i++) {
                  if (old->arr[i].type == ACTIVE) {
                        insert(hs, old->arr[i].key);
                  }
            }
            freeHashSet(old);
      }

      return hs;
}

void delete(HashSet* hs, int key) {
      int probes = 0;
      int hashVal = hash(hs, key, probes);
      while (hs->arr[hashVal].type == DELETED ||
            (hs->arr[hashVal].type == ACTIVE && hs->arr[hashVal].key != key)) {
            probes++;
            hashVal = hash(hs, key, probes);
      }
      if (hs->arr[hashVal].type == ACTIVE) {
            hs->arr[hashVal].type = DELETED;
      }
}
```