A10: Heaps

General Instructions:

1- Create a C project in eclipse called A10. Download A10.zip file and extract the following files into the project:

```
a.data.h
b.data.c
c.heap.h
d.A10_template.txt
e.A10_output.txt
f.A10_test.c
```

2- Rename: "`A10_template.txt`" to "`heap.c`". Enter your credentials on top of the file.

3- Copy the following files into your project:

```
a.queue_array.h
b.queue_array.c
c.stack_array.h
d.stack_array.c
```

4- Use the file "A10_test.c" to test your solution.

5- When you are done, you need to submit only your "heap.c" file. Do not export the project or upload a .zip file.

6- The assignment contains 8 questions, each is 1.25 points.

Overview:

In R11, you have implemented the Heap ADT. The implementation had the following features:

- *Max-heap*: The root always contains the maximum number, and each node has a value larger than its descendants.
- *Array*-Based: The implementation used arrays.

In this assignment, you are going to do two things:

- Make the Heap generic, i.e. could work as a max-heap or min-heap.
- Use a binary tree (BT) structure instead of arrays.

The heap structure is defined as the following:

```
typedef struct {
     Node *root;   //heap root
     int size;     //number of items
     char type[4]; //max or min
} Heap;
```

The *root* data member is a pointer to the first node in the binary tree. This is analogous to using the *head* for a linked list. The size data member keeps track of number of items in the heap (BT). The type is a string that is either set to “max” or “min”, to indicate whether the heap is a max-heap or a min-heap.

The structure of a node is as follows:

```
typedef struct heapNode {
     Data *data;                //data item
     struct heapNode *parent;  //pointer to parent
     struct heapNode *left;    //Pointer to left child
     struct heapNode *right;  //pointer to right child
} Node;
```

Unlike a linked list node which has only one node pointer (*next*), this node has three pointers: *parent*, *left* and *right*. Note that in a common binary tree (BT) implementation

the parent pointer is absent. However, it is introduced here to allow for top-down and bottom-up traversals of the tree (think of heapify_up and heapify_down).

We want to highlight it is not common to implement a heap using a BT, as the array-implementation is much simpler. However, this is an assignment which allows for various modifications for educational purposes.

The BT implementation of a heap is not straight-forward and would require many utility functions. Therefore, you will only do a partial implementation of the heap.

Task 1: Implementation of Heap Node:

Implement the following four functions:

```
/**
 * ------------------------------------------------------------
 * Parameters:   d -   a data item (Data*)
 *               parent- pointer to parent (Node*)
 *               left -     pointer to left child (Node*)
 *               right - pointer to right child (Node*)
 * Returns:      node -     new heap node (Node*)
 * Description:   Creates a new heap node using given data & pointers
 * Asserts:       data is not NULL
 * ------------------------------------------------------------
 */
Node* create_node(Data *d, Node *parent, Node *left, Node *right) {
     return NULL;
}
```

```
/**
 * ------------------------------------------------------
 * Parameters:   n - a node (Node*)
 * Returns:      none
 * Description:  prints the contents of a heap node
 *               invokes print_data.
 *               does not print parent, left and right
 *               if node is NULL prints <NULL Node>
 * Assert:       None
 * ------------------------------------------------------
 */
void print_node(Node *n) {
     return;
}
```

```c
/**
 * ------------------------------------------------------
 * Parameters: n -  a node (Node*)
 * Returns:    n2 - a copy node (Node*)
 * Description:Creates a copy of the given heap node
 *             The new node has a duplicate copy of data and
 *             same parent, left and right child
 * Assert:     given node is not NULL
 * ------------------------------------------------------
 */
Node* copy_node(Node *n) {
     return NULL;
}
```

```c
/**
 * ------------------------------------------------
 * Parameters:  n - a node (Node**)
 * Returns:    none
 * Description:destroys a heap node by:
 *             - destroy the encompassed data
 *             - setting parent, left and right to NULL
 *             - free the node pointer, and set it to NULL
 * Assert:     n and *n are not NULL
 * ------------------------------------------------
 */
void destroy_node(Node **n) {
     return;
}
```

Task 2: Heap Basic Functions

Implement the following four functions:

```c
/**
 * ------------------------------------------------------------
 * Parameters: type - heap type (max or min) (char*)
 * Returns:    h - new heap (Heap*)
 * Description:Creates a new heap of given type
 *             Heap root is set to NULL and size to 0
 *             if type != "max"/"min" print error msg, set to "max"
```

```
 * Asserts:    none
 * ------------------------------------------------------------
 */
Heap* create_heap(char *type) {
     return NULL;
}
```

```
/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap**)
 * Returns:    none
 * Description:destroys a heap by:
 *             - remove all items
 *             - set root to NULL
 *             - set size to 0
 *             - set type to empty string
 *             - free the heap pointer, and set it to NULL
 * Assert:     h and *h are not NULL
 * ------------------------------------------------
 */
void destroy_heap(Heap **h) {
     return;
}
```

```
/**
 * ------------------------------------------------
 * Parameters:      h - a heap (Heap*)
 * Returns:         True/False
 * Description:     check if heap is empty
 * Assert:          h is not NULL
 * ------------------------------------------------
 */
int is_empty_heap(Heap *h) {
     return 0;
}
```

```
/**
```

```
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 * Returns:     copy of data at the heap root
 * Description:Returns a duplicate copy of the data item at  root
 *             if heap is empty prints error msg, returns NULL
 * Assert:      h is not NULL
 * ------------------------------------------------
 */
Data* peek_heap(Heap *h) {
     return NULL;
}
```

Note that the insert and remove functions are left with no implementation. Keep them that way, in order for the testing file to work.

To be able to fully test these functions, you would need to complete Task 4 first, or at least provide a printing mechanism for NULL and empty heaps.

Task 3: Find Node in a Heap

Implement the following two functions:

```
/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 *              d - a data item (Data*)
 * Returns:     pointer to a node
 * Description: Returns a pointer to the node that contains given data
 *              if not found returns NULL
 * Assert:      h and are not NULL
 * ------------------------------------------------
 */
Node* find_node_heap(Heap *h, Data *d) {
     return NULL;
}

/**
 * ------------------------------------------------
 * Parameters:  n - a node (Node*)
 * Returns:     a pointer to  a node
 * Description: private helper function for find_node_heap
 *              Uses recursion.
 *              Search through given node and recurse to other nodes
 * Assert:      None
```

```
 * ------------------------------------------------
 */
Node* find_node_heap_aux(Node *n, Data *d) {
      return NULL;
}
```

Note that the two functions return a pointer to a node, not a pointer to a copy of the node.

Although traversing a BT can be done using non-recursion, i.e. through use of stacks; a recursive solution is simpler. The prototype for the auxiliary function is provided for you. If you opt to use stacks, then this is fine.

Task 4: Print a Heap

Printing a heap using the array was straight forward. Printing it using a BT-implementation would require a mechanism for tree traversal. The traversal that would make more sense for a heap is the level-order, which is done through Breadth-First-Search (BFS). A BFS is implemented using queues and not using recursion.

Implement the following function:

```
/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 * Returns:     None
 * Description: Prints contents of a heap
 *              if heap is empty prints <empty heap>
 *              if heap is NULL prints <NULL Heap>
 *              prints type and size of heap,
 *              then prints items in levelorder
 *              Uses a queue (BFS).
 *              items are separated by a space
 * Asserts:     None
 * ------------------------------------------------
 */
void print_heap(Heap *h) {
      return;
}
```

For simplicity, you do not need to have special handling for the last element. Thus, there will be an extra space at the end of printing the heap.

Task 5: Contains Heap

This function is similar to Task 3, except that you are returning True/False to indicate whether an item is in the heap or not.

You cannot use *find_node_heap* function inside your implementation.

Note that you do not need to recurse through every single node. Take use of the heap property, that in (max-heap), every node is larger than all of its descendants.

```
/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 *                  d - data to search for (Data*)
 * Returns:     True/False
 * Description: Check if a heap contains a data item
 *              recurse through all nodes to check
 *                  if any node contains the data
 *              returns True if found and False otherwise
 * Asserts:     h and d are not NULL
 * ------------------------------------------------
 */
int contains_heap(Heap *h, Data *d) {
     return 0;
}

/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 *              n - current node to check if it contains data (Node*)
 *              d - data to search for (Data*)
 * Returns:     True/False
 * Description: Private helper function for contains_heap
 *              recurse through all nodes to check if node contains data
 *              returns True if found and False otherwise
 * Asserts:     None
 * ------------------------------------------------
 */
int contains_heap_aux(Heap *h, Node *n, Data *d) {
     return 0;
}
```

Task 6: Find level of a node

This function returns the ***level*** of a given node. The root is assumed to be at level 1, and the two children of the root are at level 2, and so forth. If a node is not found, then a level of -1 is assigned to it. You can use a recursive or non-recursive solution for this function.

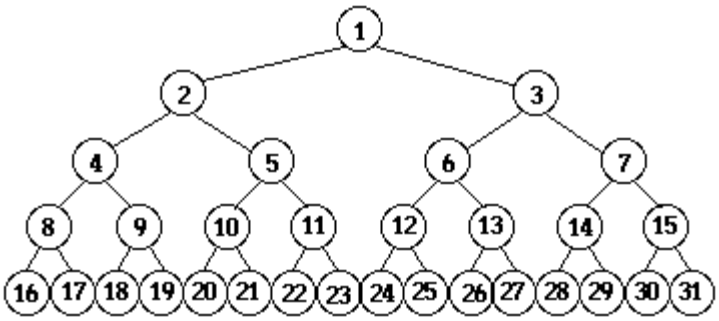
```
/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 *              node - the node to search for its level (Node*)
 * Returns:     node_level - level of "node" in the heap (int)
 * Description: Finds the level of a given node in a heap
 *              if not found returns -1
 *              Root is assumed to be at level 1
 * Asserts:     heap is not NULL
 * ------------------------------------------------
 */
int find_node_level_heap(Heap *h, Node *node) {
     return 0;
}
```

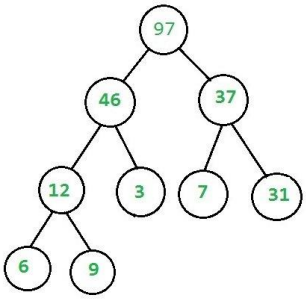
Task 7: Full Heap

This function checks if a given heap is ***full***. In tree terminology, three terms are used, complete, full and perfect. We are interested in the middle term.

A full tree/heap is structure in which each node has 0 or two children:

The following are examples of perfect heaps:





Implement the following two functions:

```
/**
 * ------------------------------------------------
 * Parameters:  root - pointer to root node (Node*)
```

```
 * Returns:     True/False
 * Description: A private helper function for is_full_heap function
 *              Uses recursion
 *              Returns True if heap is full and False otherwise
 * Asserts:     heap is not None
 * ----------------------------------------------------
 */
int is_full_heap_aux(Node *root) {
     return 0;
}

/**
 * ----------------------------------------------------
 * Parameters:  h - pointer to a heap (Heap*)
 * Returns:     True/False
 * Description: Returns True if heap is full and False otherwise
 *         Full heap is a heap in which each node has 0 or 2 children
 *              An empty heap is considered a full heap
 * Asserts:     heap is not None
 * ----------------------------------------------------
 */
int is_full_heap(Heap *h) {
     return 0;
}
```

Note that there is a very simple mathematical solution to the problem. We are not using this approach in this task.

Task 8: Max and Min of a Heap

Expand the implementation that you have provided for find_max_heap and find_min_heap in Lab10. The first function should return a copy of the maximum value, regardless of the heap being a max-heap or a min-heap. The second function should return a copy of the minimum value, regardless of the heap being a max-heap or a min-heap.

```
/**
 * ----------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 * Returns:     d - copy of data with max data
 * Description: Returns a duplicate copy of maximum item in heap
 *              steps are different depending on max/min heap
 *              if heap is empty, print error msg and return NULL
```

```
 * Asserts:     heap is not None
 * ------------------------------------------------
 */
Data* find_max_heap(Heap *h) {
     return NULL;
}

/**
 * ------------------------------------------------
 * Parameters:  h - a heap (Heap*)
 * Returns:     d - copy of data with min data
 * Description: Returns a duplicate copy of minimum item in heap
 *              steps are different depending on max/min heap
 *              if heap is empty, print error msg and return NULL
 * Asserts:     heap is not None
 * ------------------------------------------------
 */
Data* find_min_heap(Heap *h) {
     return NULL;
}
```

Using recursion is the preferred method for both functions. Since there are multiple ways to apply the recursion, the function prototypes for the helper functions are not provided.