C Final Test
Heapsort algorithm

Monday 12th June 2017, 10:00

THREE HOURS

(including 10 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time, log in using your username as **both** your username and password.

The maximum mark is 30.

Credit will be awarded throughout for clarity, conciseness and *useful* commenting.

Important note: THREE MARKS will be deducted from solutions that do not compile or with memory leaks. Comment out any code that does not compile before you leave. Commented-out code will not be marked.

1 Heapsort

In this test you will implement a comparison-based sorting algorithm called Heapsort. Unlike competing algorithms such as Quicksort, which has worst-case complexity $O(n^2)$, the worst-case complexity of Heapsort is $O(n\,log(n))$. This makes Heapsort popular in embedded systems with real-time constraints and systems concerned with security, e.g. the Linux kernel.

2 Binary heap

A binary heap (sometimes abbreviated here to just 'heap') is a data structure that can be viewed in the form of a complete binary tree, which means that all levels of the tree, with an exception of the deepest level (the last one), are fully filled. If the last level of the tree is not complete, the nodes of that level appear in the leftmost leaf positions. Figure 1 shows an example of a binary heap with 11 nodes, four of which are leftmost leaves in the last level. In this test, nodes will be numbered from 1 and the items stored in the nodes (the *key*) will be characters, as in Figure 1.

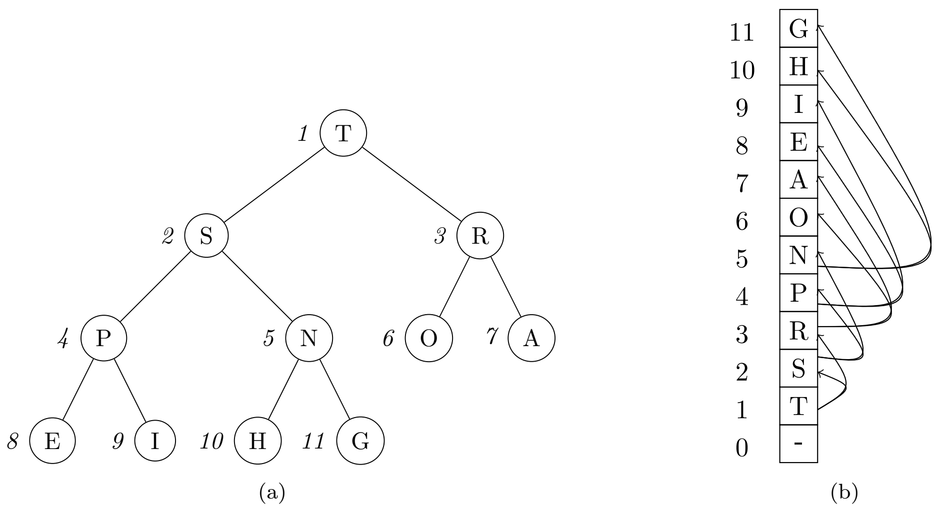

Figure 1: Example of binary heap in a form of (a) semi-complete binary tree representation and (b) array representation. The item inside each node on the tree is the key stored at that node. The number outside each node is the corresponding index in the array representation. Notice, that the index 0 in the array representation is not used, so a heap of size N is represented by an array of length $N + 1$.

A binary heap by definition satisfies two properties:

- **Shape property**: as mentioned, a binary heap is usually a complete binary tree. However, if the tree is incomplete, the nodes are filled from *left* to *right*. The binary heap can also by represented by an array, where the parent node at index k is in $\lfloor k/2 \rfloor$ and the two children of a parent node in index k are in index *2k* (left child) and *2k+1* (right child) (see Figure 1 (b)). In this test you will represent binary heaps using arrays as described above.
- **Heap property**: the key is stored in each node following some *total order*. In this test, since the nodes' keys are `chars`, the *total order* will follow the alphabetical order. In this test, you will implement a so called *max-heap* because the heapsort sorting algorithm uses max-heaps. In a *max-heap* the key of a node is *always greater than or equal* the keys in the

children (left and right) nodes; for every node i, other than the root node, its stored key is at most the value of its parent. Therefore, the largest element in the max-heap is stored at the root (i.e. index 1 in the corresponding array).[1]

In the worst case, the basic operations in a heap (such as insertion or deletion of elements) take time proportional to the depth of the heap, i.e. $O(log(n))$. In the first part of this test (PartI), you will work on the implementation of the following operations involving the heap data structure:

- *max-heapify*: restores the heap order by enforcing the nodes to maintain the max-heap property, i.e. that the key of a node is always greater than or equal the keys of its child nodes.
- *max heap construction*: builds, a max-heap tree from an initial arbitrary binary tree order, using the array representation described above.
- *heapsort*: repeatedly removes the largest item in the heap (array) and stores it back in the array at the index corresponding to the number of elements currently in the heap. As the heap shrinks, this index is therefore reduced so that the final element (the smallest element in the original heap) ends up at index 1. In the worst case, sorting in this way will take $O(n\,log(n))$ time.
- *insert element in max-heap*: adds a key to the heap.

The Heapsort algorithm is broken into two parts: the first part is *heap construction*, where an initial array of characters keys is re-organised using *max-heapify* and the second part where all the items are pulled out of the heap in a decreasing order to build a sorted array as described above. The *heap construction* takes time proportional to $O(n\,log(n))$. It works by going through the array from left to right applying *max-heapify* at each index as described below.

2.1 The Max-heap property

In order to maintain the *max-heap property*, the function `max_heapify` is used. Its input arguments are the heap array, `heap`, an index `i` into the array and the size of the heap `heap_size`. When `max_heapify` is called, it assumes the precondition that the binary trees rooted at node `i`'s children, `left_child(i)` and `right_child(i)`, satisfy the *max-heap* property. However `heap[i]` might be smaller than its children, therefore breaking the *max-heap* property, in which case `max_heapify` is called to restore the invariant. Figure 2 describes the whole process.

2.2 Building a max-heap

`build_max_heap` consists of applying `max_heapify`, in a bottom-up manner, to convert the initial array into a max-heap. Notice in Figure 1 (b), elements 1 to 5 are parent nodes and 6 to 11 are leaves. In general, `heap[`$(1 + ... + (\lfloor heapsize/2 \rfloor)$`]` are parent nodes and `heap[`$(\lfloor heapsize/2 \rfloor + 1)..heapsize$`]` are all leaves of the tree (i.e children) and each one of them is a one-element heap to begin with and therefore a root of a max-heap. `build_max_heap` uses a decrementing loop through the non-leaves nodes to go through the rest of the nodes of the tree running `max_heapify` on each one of them to enforce *max-heap* property. Figure 3 describes the whole process.
To illustrate that `build_max_heap` works, the above loop invariant i.e. each node $(i + 1, i + 2, ..., heapsize$ where $i = \lfloor heapsize/2 \rfloor)$ in the `heap` array is a root of a max-heap is applied. Looking at the `heap` array (Figure 3 (a)) the index of a child node n in the array is higher than the index of its parent node. Applying the above loop invariant, this means that the child

[1] Introduction to algorithms. C. Leiserson and R. stein. (third Ed).

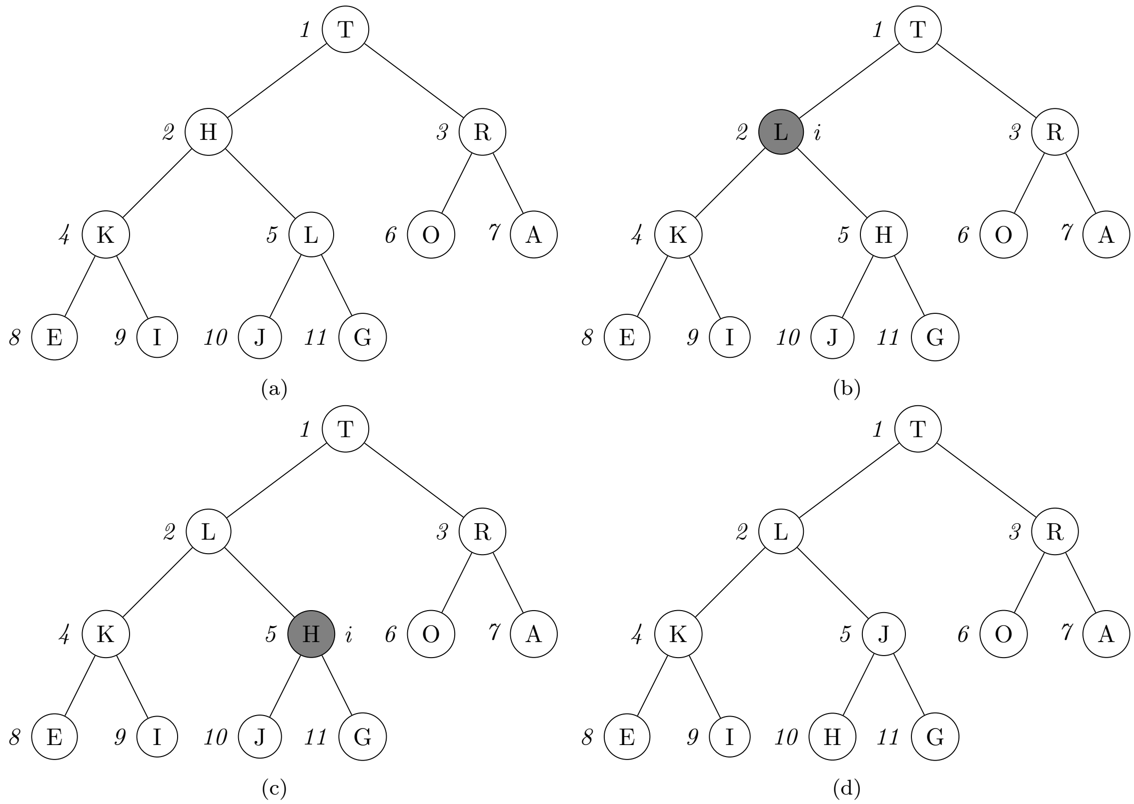

Figure 2: (a) shows a binary tree where the node 2 of the `heap` array is breaking the *max-heap* property as the parent node is not bigger than the child nodes 4 (`left_child`) and 5 (`right_child`). The figure assumes the initial call is `max_heapify(heap,2,11)` where the `size_heap` is 11. The max-heap property is restored for node 2 by swapping `heap[2]` for `heap[5]` see (b). But now, node 5 of the heap array is breaking the max-heap property as it is smaller than child node 10 (`left_child`). The recursive call to `max_heapify(heap,5,11)` is shown in (c). It restores the *max-heap* property by swapping `heap[5]` for `heap[10]` as shown in (d). Further recursive calls in `max_heapify(heap,10,11)` will not change the data structure as all 1-node elements are root of a max-heap.

in `heap[`$(\lfloor heapsize/2 \rfloor + 1)..heapsize$`]` is a root of trivial, one-element, max-heap. This is the condition assumed when you call `max_heapify` at some index `i` to make the node `i` a max-heap root. If `max_heapify` preserves the *max-heap* property, i.e. that nodes in `heap[`$i..heapsize$`]` are all roots of max-heaps, then **decrementing** n in the loop in `build_max_heap` re-establishes the loop invariant for the next iteration.

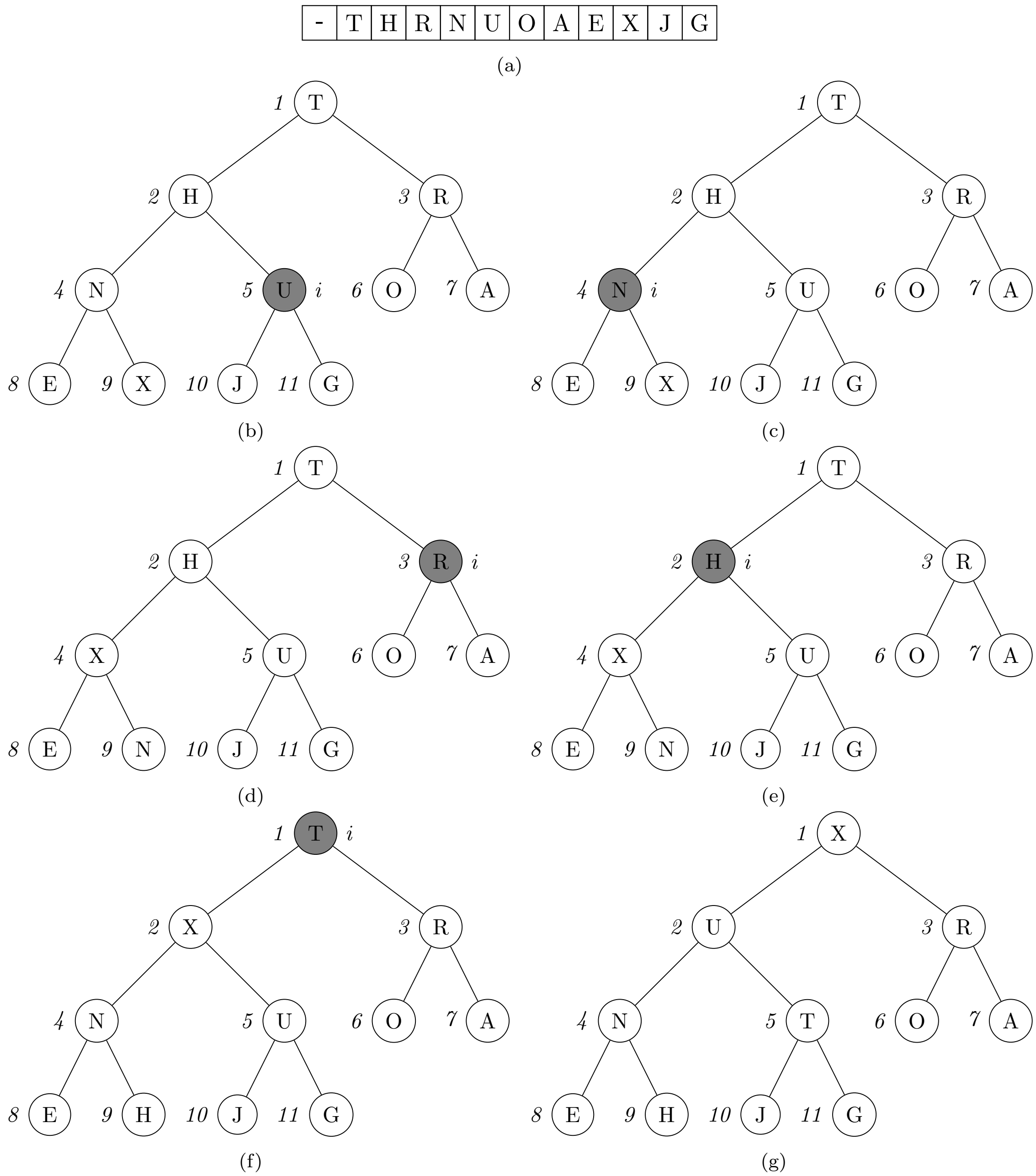

Figure 3: (a) shows the `heap` array with a `heap_size` of 11-elements, this is the data structure input in `build_max_heap` before starting to call `max_heapify` from bottom-up manner. (b) shows a binary tree representation where the loop at index i refers to node 5 (shaded) when `max_heapify` is called. (c) shows a binary tree representation after applying `max_heapify` to node 5. The loop index i for the next iteration refers to node 4 (shaded). (d)-(f) shows the subsequent iterations of the loop in `build_max_heap`. Notice that whenever `max_heapify` is called on a node, the two subsequent trees of that node are both max-heaps by construction. (g) presents the result of the data structure as max-heap after applying `build_max_heap`.

2.3 Heapsort algorithm

The heapsort algorithm starts by building a max-heap on a `heap` array of size `heap_size` by using `build_max_heap`. The data structure after applying `build_max_heap` has the maximum element of the `heap` array in the first index `heap[1]` (Figure 3 (g)). Therefore, this element can be stored in the final position of the `heap` array by *exchanging or swapping* `heap[1]` with `heap[heap_size]`. In order to restore the max-heap property `max_heapify` is then applied to the resulting tree which now has fever elements. The process then repeats until there are no more elements on the heap. Figure 4 shows the whole process.

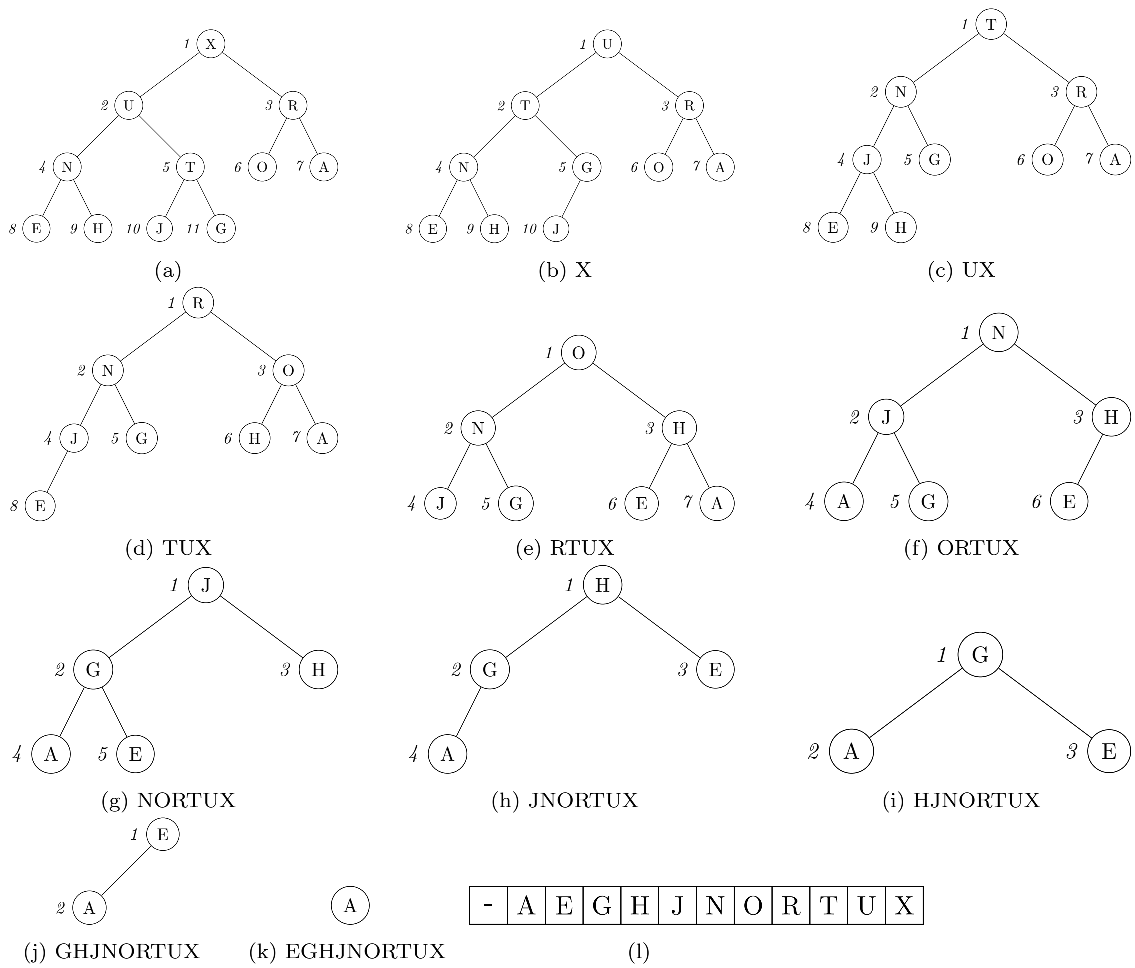

Figure 4: This figure illustrates the heapsort algorithm. (a) shows the max-heap data structure after applying `build_max_heap`. (b)-(k) show the heapsort algorithm steps when removing the root node and applying `max_heapify` to maintain the *max-heap* property. Notice that the number of nodes in the heap decreases at each step. (l) shows the `heap` array sorted.

3 What to do

The test consists of two parts and you are provided with the files for each part. In part I (see `PartI` directory), you are provided with a main program file `heapsort.c`, a header file `binaryheap.h` and a program file `binaryheap.c`. The `binaryheap.c` file is where you will write your code while the file `heapsort.c` contains a definition of a `main` function which will call the functions you are required to complete during part I of this test. At present the functions in `binaryheap.c` are nothing more than stubs; you should fill out each as you write and test your functions.

The file `binaryheap.h` contains prototypes of all the functions you are expected to write. Do *not* modify the definitions in `binaryheap.h` as this will impede our ability to evaluate and test your code.

You may also define and use appropriate auxiliary functions in `binaryheap.c` and `binaryheap.h`. These should be appropriately scoped and not be visible outside the module in which they are defined. You may compile your code using the provided `Makefile`. By default your code will be compiled with debugging symbols (via `gcc`'s `-g` option) so you will be able to debug your code with a capable debugger: `gdb`, `cgdb`, `clion`, or `eclipse` for example.

The `Makefile` instructs `make` to build an executable program called `heapsort`. You may assume that all inputs to the program are valid and, for example, that any text to be sorted contains characters present in the supplied binary tree.

The `heapsort` program should accept a string provided by the user and print the elements on the `heap` array, then the elements on the `heap` array after applying `build_max_heap` and then the elements in the `heap` array after applying the `heapsort` algorithm. For example, given the input string THRNUOAEXJG your program should print out something like:

```
T H R N U O A E X J G
1 2 3 4 5 6 7 8 9 10 11
X U R N T O A E H J G
9 5 3 4 1 6 7 8 2 10 11
A E G H J N O R T U X
7 8 11 2 10 4 6 3 1 5 9
```

In Part II, you need to work on the `program.c` file provided (see `PartII` directory). The file `program.c` contains a program that you need to debug and find all the possible bugs (if any) at compile time or run-time.

4 Given `structs` and `typedefs`

For the purpose of this test, the `heap` array will contain `struct` elements which contain a key (pointer of a char) and an index per each node. The file `binaryheap.h` defines the following `struct` and `typedef` for defining the elements on the `heap` array:

```
typedef struct node_heap_t node_heap;

struct node_heap_t {
 char *key;
 int position;
};
```

Do not modify the `struct` and `typedef` provided for you.

Part I – Heapsort

Thirteen functions should be implemented in `heapsort.c`:

1. Write a function:

   ```
   node_heap *allocate_node_heap(void);
   ```

   which allocates memory for a `node_heap` type and returns a pointer to this new node.

   [3 marks]

2. Write a function:

   ```
   void initial_heap(node_heap **heap, char* sequence);
   ```

   which traverses the given string sequence. For each character in the null-terminated sequence, it creates and allocates memory for another `node_heap` element and adds it into the `heap` array. Then, it makes the key of the new `node_heap` to point to the character and also assigns to the `node_heap`'s position the position of the character in the sequence. Hint: you could use `allocate_node_heap`.

   [3 marks]

3. Write a function:

   ```
   void print_elem_heap(node_heap **heap, int length);
   ```

   which prints two lines with the `key` and `index` of each `node_heap` element in the `heap` array. The format to print the information on a `heap` array of elements is as follows:

   [2 marks]

   ```
   T H R N U O A E X J G
   1 2 3 4 5 6 7 8 9 10 11
   ```

   Notice that there is only a single space between `key` and `position` values.

4. Write a function:

   ```
   int parent(int index);
   ```

   which returns the parent node's index in the `heap` array given the index passed as argument.

   [0.5 marks]

5. Write a function:

   ```
   int left_child(int index);
   ```

   which returns the left child node's index in the `heap` array given the parent's index.

   [0.5 marks]

6. Write a function:

```
int right_child(int index);
```

which returns the right child node's index in the `heap` array given parent's index.

[0.5 marks]

7. Write a function:

```
void swap(node_heap *node1, node_heap *node2);
```

which swaps the information in `node_heap *node1` to `*node2` and vice versa.

[1.5 marks]

8. Write a function:

```
void max_heapify(node_heap **heap, int current, int heap_size);
```

which ensures that the `node_heap` element in the `current` position of the `heap` array satisfies *max-heapify* property. If it does not, then `swap` the current node with the largest node amongst its children and apply `max_heapify` again onto the largest node to ensure the max-heap is accomplished in the subtree.

[5.5 marks]

9. Write a function:

```
void build_max_heap(node_heap **heap, int heap_size);
```

which traverses the `heap` array in a bottom-up manner, enforcing the property that all the elements in the `heap` array form a max-heap. Hint: follow the loop invariant 2.2. Also, remember that `heap[0]` is unused.

[1 marks]

10. Write a function:

```
void heapsort(node_heap **heap, int length);
```

which initially assumes that the `length` of the `heap` array is the same as the `heap_size`. It starts with a `build_max_heap` and then traverses the `heap` array using a decrementing loop. Each time, it swaps the largest element in the `heap` array with the element at index given by `heap_size`, which in general has the effect of constructing a tree that does not satisfy the *max-heap* property. Then it decrements the `heap_size` and restores the *max-heap* property.

[2.5 marks]

11. Write a function:

```
void free_node(node_heap *node);
```

which frees the memory allocated for a `node_heap.`

[1 marks]

12. Write a function:

```
void free_heap(node_heap **heap, int length);
```

which traverses `heap` array and frees every `node_heap` element of the array. Hint: use `free_node`.

[1 marks]

13. Write a function:

```
int main(int argc, char **argv);
```

which given a string creates an array `heap` of pointers to `node_heap` elements. You may assume that the string passed to `main` will contain no more than 20 characters. This should initialise the `heap` array with the string sequence also provided by the user. Then, print the elements on the heap. Build the max heap and print again the results. Then, apply the heapsort algorithm and print the results and terminate successfully the program. Below, there is an example of the outputs you should get given a string sequence:

> ./heapsort THRNUOAEXJG
T H R N U O A E X J G
1 2 3 4 5 6 7 8 9 10 11
X U R N T O A E H J G
9 5 3 4 1 6 7 8 2 10 11
A E G H J N O R T U X
7 8 11 2 10 4 6 3 1 5 9

> ./heapsort SORTEXAMPLE
S O R T E X A M P L E
1 2 3 4 5 6 7 8 9 10 11
X T S P L R A M O E E
6 4 1 9 10 3 7 8 2 5 11
A E E L M O P R S T X
7 11 5 10 8 2 9 3 1 4 6

[3.5 marks]

Part II – Inspecting and debugging a program

1. You are provided with an application called `program`, which allocates in memory a string s and returns another string containing only the unique characters of s. Determine why `program` is not behaving properly. You should list all the possible bugs (if any) in `program.c` and proposed a solution (if any) per bug. If you spot a bug (including segmentation faults), comment out the line of code that contains the bug and write any proposed solution on the same file `program.c`, make sure that you comment your solution to help us evaluate your proposed solution. HINT: use Valgrind. If `program` works correctly, it should output the following:

   > ./program
   The initial word is: attack
   Derived lookup table: atck

   [4.5 marks]