Project Five: Texture Packing

Date: 2020-05-04

Chapter 1: Introduction

1.1 Problem Description

Texture Packing is a strip-packing problem and is a common topic explored in the area of Approximation Algorithms. It is classified as a NP-complete problem, where the decision model asks if a set of items would into a pre-determined number of levels. Unlike the *Bin Packing problem*, there are two parameters (width and height) instead of one (bin capacity) are taken into calculation.

Given a set of N rectangles with dimensions width w_i and height h_i where $i = 1, 2, \ldots, n$, we are expected to pack them into a larger shape with a pre-specified width W. We want to pack as many rectangles as possible into the larger one with the objective of minimizing the height, in polynomial time. Here, we will demonstrate how this can accomplished by using a the *Next-Fit Decreasing Height (NFDH)* approximation algorithm.

1.2 Input and Output Specification

- Input
  The program first receives, in one line, two positive integers W and N, where W is the width of the texture atlas that the images will be packed into and N is the number of images to be packed. The program then receives N lines of input. Each line specifies an image and contains two positive integers h and w, the pixel height and width of the image, respectively.

- Output
  The program first outputs the total area used by the texture atlas. It then outputs N lines, specifying the positions of the images in the texture atlas. Each line contains two positive integers x and y, the x and y position of the bottom-left pixel of the image in the texture atlas, respectively. The positions are outputted in the same order as the input sizes are input; that is, the i^{th} position in output corresponds to the i^{th} image in input.

For examples of expected inputs and outputs, please refer to the appendix (*Examples of Input & Output*) on page 10.

Chapter 2: Algorithm Specification

2.1 Data Structures

- Min-Heap
  A min heap is used in the implementation of heap sort, and the resulting sequence is stored into an array. A struct was used to represent this data structure and had attributes each corresponding to the nodes, number of nodes in the heap, and heap capacity.

```
struct minheap_struct{
    int *nodes;
    int size;
    int capacity
};
```

Note that the node does not store the height of each inputted rectangle, but rather the ***address*** of each shape.

2.2 Algorithm Specifications

- Next-Fit Decreasing Height (NFDH) Algorithm: The Main Idea
  The NFDH algorithm is an off-line algorithm, where all input is considered before an output is produced. Every inputted rectangle is sorted by their height in decreasing order, where the tallest gets dealt with first and is placed in the left of the texture atlas. A *next-fit* approach is used to pack the shapes, where a rectangle is packed in if and only if it does not exceed the width requirement. If the next rectangle to be packed goes over the width, it will be placed above the previous shape and justified left. This algorithm tends to pack the rectangles level-by-level, and earlier levels cannot be accessed.

- Sorting the Input – `buildheap()`, `sort()`, `pop()`
  To sort the inputted rectangles in decreasing order by their height, we use heap sort. First, we build a min-heap using the address of each rectangle as the node's key. We then pop the root, place it into the array `node[]` and heapify accordingly. The pseudocode for heap sort and related heap operations are trivial and have been omitted.

- Packing the Rectangles Together – `nfdhPack()`
  The pseudocode uses both rectangles and images interchangeably. For reference purposes, images will refer to the inputted set of rectangles.

```
int nfdhPack(){
    int lvl_height; //lvl_height stores the height of each level
    int used_width; //used_width stores width of space already occupied in each level

    /*empty level implies lvl_height = 0 and used_width = 0.*/

        /*Initializing pointer variables;
    current_image is initialized to the rectangle with the greatest height*/
    current_lvl := 0; //Points to the top or “current” level
    current_image := images[0]; //Points to the current rectangle to be packed
    current_y := 0; //Contains the current y-coordinate of the rectangles

        /*Initialize variables to hold coordinates of the current rectangle being worked with;
        The coordinates are defaulted to the bottom left (0,0)*/
    x_pos[current_image] := 0;
    y_pos[current_image] := 0;

    /*Initialize the attributes of the first level with the height and width of
    the first rectangle*/
    lvl_height[0] := image_height[current_image]; //stores the height of each level
    used_width[0] := image_width[current_image]; //stores width of space occupied per level
    for each rectangle to be packed:
        current_image := current rectangle to be packed;

        /*Check if the current rectangle can be packed into the texture atlas*/
        if (width of the level of current_image > remaining width of texture atlas
        || height of current level < height of current rectangle):
            current_y += add the height of the previous level
            set current_lvl to the next level;
            lvl_height[current_lvl] := height of the current rectangle;

                /*Update coordinates of packed shape in texture atlas*/
            x_pos[current_image] := left-justified position of current rectangle and level;
            y_pos[current_image] := justified position of current rectangle and level;
            update remaining amount of width in current level;

    current_y := current y-coordinate + height of the current level
    return current_y;   //Total height of the texture atlas is returned
}
```

The above pseudocode is an implementation of the NFDH approximation algorithm. As described in "*Next-Fit Decreasing Height (NFDH) Algorithm: The Main Idea"*, the idea is to take a pre-sorted set of rectangles and pack them into a texture atlas, creating a new level each time the specified width is exceeded. Our implementation first takes the rectangle with the greatest height and initializes the current height and width of the first level. For every rectangle after the first rectangle, the algorithm first checks to see whether the current level is too short or if there is too little unused width remaining to fit the shape. If so, the rectangle gets packed into the atlas at the next level. This in turn makes the next level, updates the y-coordinate pointer of the current level and the height of the new level. Else, we can pack the next rectangle in the current level. Regardless of whether the rectangle gets packed into a new level, the x and y coordinates, and the current width of the current level gets updated. Finally, the total height of the texture atlas gets returned.

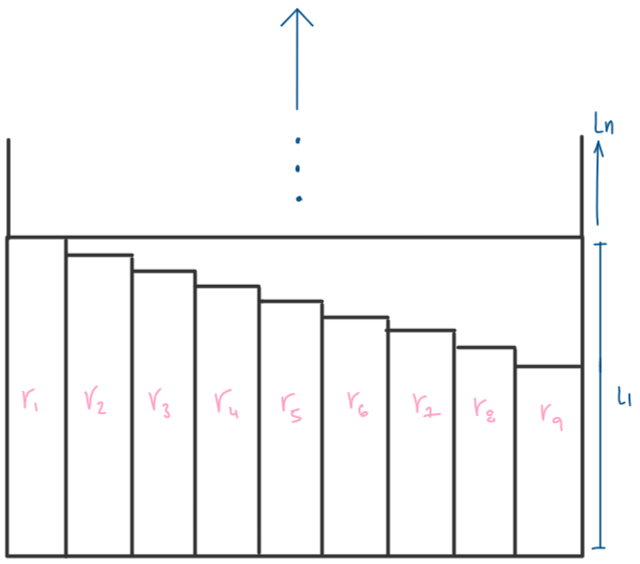
Chapter 3: Testing Results

3.1 Test Cases

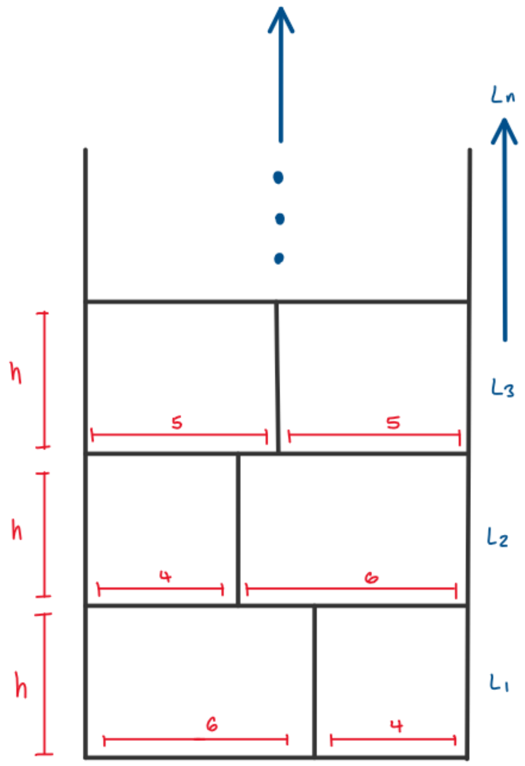
Let N be the size of the set of the rectangles to pack, W_L be the width of each level, w_i and h_i respectively be the width and height of each rectangle r_i where , $i = 1, 2, \dots, N$.

- *Random Case:* Heights and widths of r_i are randomized, ranging from $[0, W_L]$
- *Trivial Case:* All r_i have the same width and height of W_L .
$$w_i = h_i = W_L \ , i = 1, 2, \dots, N$$
- *Diagonal Case:* All r_i have the same width and height.
$$w_i = h_i \ , i = 1, 2, \dots, N$$
- *Mismatched Diagonal Case:* Each r_i has $w_i = W_L - h_i$.

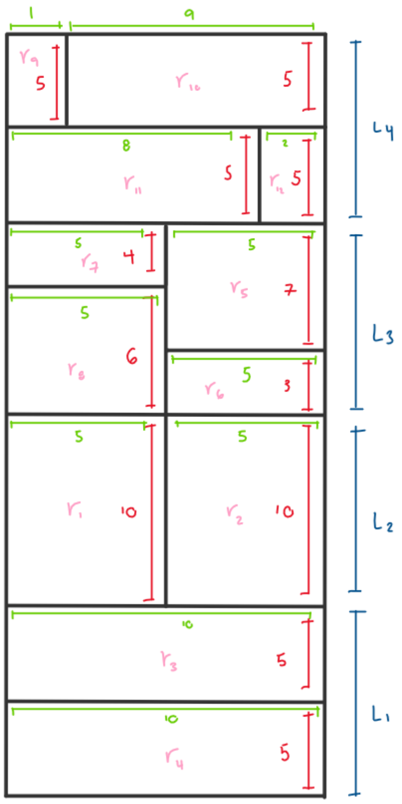
- *Best Case:* Smallest size of $N = 10$, where all 10 r_i can fit together in the same level.



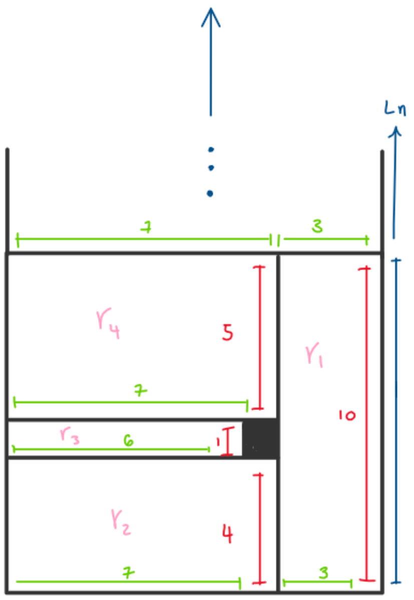
- *Worst Case:* Largest size: $N = 10000$, where no rectangle can fit together in the same level.
- *All-Pairs Case:* Assume $W_L = 10$. All r_i have a fixed w_i, such that $4 \leq w_i \leq 6$ and the same h_i where $h_i \in \boldsymbol{Z}^+$. The optimal solution would have no whitespace, and every level should look like one of L_1, L_2, or L_3 .



- *Filled (No Gaps) Case:* In an optimal solution, no level would have any whitespace, and would look like either L_1, L_2, L_3 or L_4. Assume $W_L = 10$.



- *Filled (With Gaps) Case:* In an optimal solution, every level would have a 1×1 unit of whitespace, depicted below. Assume $W_L = 10$.



**The black box represents a unit of whitespace.*

3.2 Correctness Testing

We know that for an optimal case, our *filled*, *no gaps case* should take up exactly 4 levels for every series of 12 rectangles, and our *filled*, *with gaps case* should take up exactly 1 level for every series of 4 rectangles. We also know that our *trivial case* and *worst case* should take up exactly as many levels as the number of rectangles in the case, and that our best case should take up exactly 1 level. As well, we know that our *all-pairs case* has an optimal solution in which there is no whitespace, and given an even N will have exactly $\left\lfloor \frac{N}{3} \right\rfloor + \left\lceil \frac{N}{3} \right\rceil$ levels, which for $N = 1000$ will be 500 levels. We test these cases in the table below.

Test case description	Number of rectangles (N)	# of levels used	Expected # of levels used
Filled (no gaps) case	60	22	20
Filled (with gaps) case	20	7	5
Trivial Case	100	100	100
Worst Case	10000	10000	10000
All-Pairs Case	1000	500	500

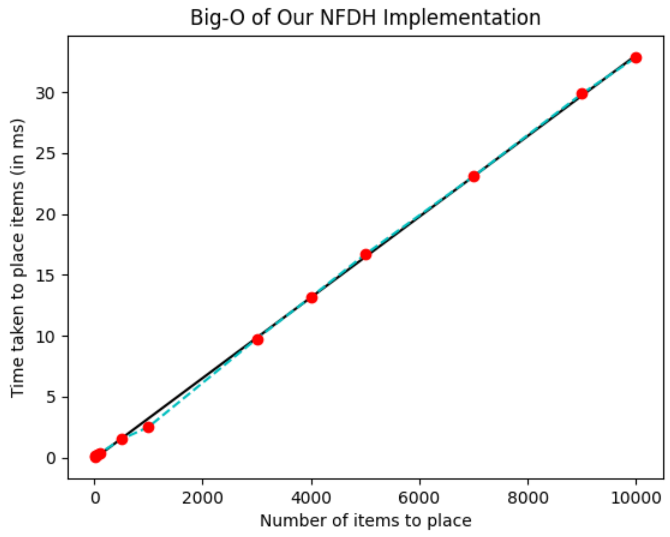
As we can see, all the amounts of levels used are either the optimal case, or within the approximation ratio of 2 ($NFDH(I) \leq 2 \cdot OPT(I) + 1$, where I is an arbitrary set of rectangles) for the optimal case. Therefore, we know our algorithm is correct!

3.3 Performance Testing

Test case description	Number of items (N)	Time taken (in ms)
Best Case	10	0.088
Filled (with gaps) Case	20	0.125
Filled (w/ no gaps) Case	60	0.22
Trivial Case	100	0.3287
Random Case	500	1.4937
All-Pairs Case	1000	2.491
Diagonal Case	3000	9.7453
Mismatched Diagonals Case	4000	13.167
Random Case	5000	16.6687
Random Case	7000	23.08
Random Case	9000	29.8653
Worst Case	10000	32.853

*Time taken is accurate to 4 decimal places.

The graph made from these test cases is show below:



In this graph, the black line is the line of best fit, done via linear regression, the red dots are the points that we tested, and those points are connected with a dotted cyan line. The cyan line only differs slightly from the best fit line, while the best fit line is close or equal to $O(N)$. This is because for smaller N, $O(NlogN)$ and $O(N)$ are very close, and as seen in the dip at the beginning of the graph, for very small N, $O(NlogN)$ can be even faster. As the number of rectangles increase, however, the cyan line moves ever so slightly above the black line, showing that for larger N the program does indeed break the $O(N)$ upper bound. We believe the reason we cannot see a significant breaking of this bound is only due to the upper limit of test case sizes given in the question, which is 10000.

Running the worst case on my personal computer, takes up to 35 seconds in real time and since the real bound of the program is more so gcc's `scanf` implementation than any part of our algorithm (which was not counted for our tests), so trying to find the N for which the algorithm would more cleanly break an $O(N)$ upper bound would be impractical.

Chapter 4: Analysis and Comments

4.1 Time Complexity

As stated above in *Chapter 3*, our testing and analysis tends to a runtime of $O(NlogN)$. For a more generalized analysis, we break apart our algorithm into three key components – `buildHeap()`, `sort()` and `nfdhPack()`. The cost of pointers and malloc/calloc calls are omitted in this calculation.

The algorithm to construct a min-heap of the rectangles according to their heights, `buildHeap()` takes $O(N)$ time to do so. To sort the heap in decreasing order by their heights, heap sort (`sort()`) is used and finishes the task in $O(NlogN)$ time. Lastly, the running time of the NFDH algorithm (`nfdhPack()`) is bounded by $O(N)$, since the rectangles in the set are already pre-sorted.

$$\therefore\ T(N) = O(N) + O(NlogN) + O(N) = O(NlogN)$$

Thus, the running time of our algorithm is $O(NlogN)$.

4.2 Space Complexity

The space complexity for our approximation algorithm is $O(N)$. Only one-dimensional arrays are used to keep track of the input, the x and y coordinates, heights of each level in the texture atlas and so on. Furthermore, the min-heap data structure uses an array implementation making it extremely efficient. Dynamic memory allocation is also utilized, which contributes to the effective use of space in the program.

$$\therefore\ S(N) = O(N)$$

4.3 Approximation Analysis

The following analysis was done based on the background given by [1]. As stated earlier, the texture-packing problem is classified as NP-complete. We can use approximation algorithms as heuristic method to find the near-optimal solution for this problem and use the approximation ratio to examine the difference from optimal.

Let I denote an arbitrary set of rectangles, where each rectangle has width of 1. Let A denote the approximation algorithm, and $A(I)$ denote the actual height used by A. Finally, let $OPT(I)$ denote the optimal algorithm where the smallest possible level height which the rectangles in I can be packed. We will prove the asymptotic performance bound of A in the form of

$$A(I) \leq \beta \cdot OPT(I) + \gamma,$$

where β is a constant that represents the approximation ratio and γ as an additional constant of height.

If the height and width of each rectangle is normalized to be no more than 1, then

$NFDH(I) \leq 2 \cdot OPT(I) + 1$ from Theorem 1 of [1],

for all arbitrary instances of I. Thus, the asymptomatic performance bound of 2 is tight for all I sorted in decreasing height.

Conclusion

Comparing the NFDH approximation algorithm to other strip-packing algorithms, it does not necessarily have the best performance or approximation ratio. Despite it being relatively naïve, it efficiently generates usable packings for almost all cases. However, the algorithm has clear flaws.

The algorithm tends to leave a large amount of empty space on the right side of each level. Algorithms exist which capitalize on this flaw, such as *Baker's Up-Down algorithm*. Furthermore, in the case where one image is significantly taller than all the others, the level sorting approach leads to wasted space because a single image cannot span multiple levels. This problem can be solved by a wholistic packing algorithm, one that does not pack into levels.

When seeking a packing of images that reduces area, the problem as posed in this project is insufficient. The width of the final packing is restricted to the value given in the input. It is possible that no truly efficient packing using this width can exist for a given set of images. In future research, two-dimensional packing algorithms that can produce packings with a range of widths must be considered.

Appendix

- Examples of Input & Output

```
 **Input 0:**


10 10
4 5
1 1
3 2
1 6
2 9
1 3
2 2
2 4
5 3
8 1
**Output 0:**

area=130

(3, 0)
(8, 12)
(7, 9)
(2, 0)
(0, 0)
(9, 0)
(5, 9)
(7, 0)
(0, 9)
(0, 12)
```

```
**Input 1:**

25 17
1 9
2 12
3 15
8 6
9 14
9 6
2 2
1 3
2 7
5 1
```

```
10 9
8 25
1 12
4 6
5 9
12 5
7 18

**Output 1:**

area=1375

(5, 39)
(10, 25)
(15, 0)
(0, 48)
(0, 25)
(12, 39)
(21, 48)
(20, 48)
(6, 39)
(0, 54)
(12, 25)
(0, 0)
(9, 25)
(8, 39)
(0, 39)
(8, 48)
(8, 0)
```

```
**Input 2:**

1 3
5 2
4 3
1 6

**Output 2:**

Invalid Input
```

- Source Code: *project5nfdh.c*

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>

//algorithm adapted from the nfdh algorithm described at: https://cgi.csc.liv.ac.uk/~epa/surveyhtml.html#b
ib.1

struct minheap_struct{  //min_heap structure is used to implement heap sort
    int *nodes; //stores the nodes in the min_heap. the 0th element in the array should be a sentinel node
    int size;   //stores the number of nodes in the heap
    int capacity;   //stores the maximum number of nodes the heap can store, based on the memory allocated
};

typedef struct minheap_struct *minheap;

int texture_width, num_images;  //texture_width stores the given width of the texture atlas. num_images st
ores the number of images to be packed
int *image_height, *image_width, *x_pos, *y_pos;    //image_height and image_width store the heights and w
idths of the images, respectively
                                                    //x_pos and y_pos store the x and y coordinates, respe
ctively, of the bottom left pixel in the image

minheap buildheap(); //O(N) algorithm to build a min heap from the images according to their heights, retu
rns a heap
int pop(minheap h);   //removes the top item from the heap h, maintains the heap structure, and returns th
e removed item
int* sort(minheap h); //iterativey pops items from the heap h until it is empty, returns an array containi
ng the items in decreasing order of their heights
int isEmpty(minheap h); //checks if the heap h is empty. returns 1 if true, 0 if false

int nfdhPack(int* images);   //packs images in the order given by the parameter images. writes the positio
n of images in the NFDH packing to x_pos, y_pos, returns the height of the packing

int main(){
    printf("Input:\n");
    scanf("%d %d", &texture_width, &num_images);

    clock_t begin = clock(); //Time at beginning (after the scanf)

    image_height=(int*)malloc(sizeof(int)*num_images);  //stores the heights of image
    image_width=(int*)malloc(sizeof(int)*num_images);   //stores the widths of the image

    x_pos=(int*)malloc(sizeof(int)*num_images);    //stores the x_pos of the bottom left pixel of each ima
ge
    y_pos=(int*)malloc(sizeof(int)*num_images);    //stores the y_pos of the bottom left pixel of each ima
ge
```

```c
    int error_flag=0;
    for(int i=0;i<num_images;i++){  //for all i images
        scanf("%d %d", image_width+i, image_height+i);    //read in the height and width of the image
        if(image_width[i]<1 || image_height[i]<1 || image_width[i]>texture_width){   //if the image has ne
gative or zero dimensions or if the image is wider than the texture atlas
            error_flag=1;   //catch the error
        }
    }

    if(error_flag){ //an error was caught
        printf("Invalid Input\n");
        return 1;   //end the program because the input is invalid, return 1 to flag error
    }

    minheap h=buildheap();   //builds a min heap of the images
    int* sortedImages=sort(h);   //sorts the images in decreasing order of their heights and stores them i
n the int array sorted

    free(h);    //deallocates h as it is no longer needed

    int area=texture_width*nfdhPack(sortedImages);  //calls the 2d packing algorithm

    printf("\nOutput:\n");
    printf("area=%d\n",area);
    for(int i=0;i<num_images;i++)   printf("(%d, %d)\n", x_pos[i], y_pos[i]);   //output the (x,y) coordin
ates of the images in the texture atlas

    clock_t end = clock(); //Time at end
    double elapsed_time = ((double)(end - begin) / CLOCKS_PER_SEC) * 1000; //Time taken = time at beginnin
g - time at end; Multiply by 10^3 to get milliseconds
    printf("\nTime taken by program was %f milliseconds\n", elapsed_time);

    return 0;   //end of program, return 1, no errors
}

int nfdhPack(int* images){
    int* lvl_height=(int*)calloc(sizeof(int), num_images);    //stores the height of each level. an empty
level has height 0
    int* used_width=(int*)calloc(sizeof(int), num_images);    //stores the width of space already occupied
 in each level. and empty level has 0 width.

    int current_lvl=0;  //used as pointer to the top level
    int current_image=images[0];    //used as pointer to the current image. initialized as image with grea
test height
```

```
    int current_y=0;    //used to set the y coordinates of images, initialized to 0

    x_pos[current_image]=0; //set the x coordinate of the first image, by default it goes in the bottom left
    y_pos[current_image]=0; //set the y coordinate of the first image
    lvl_height[0]=image_height[current_image];  //set the height of the first level, same as height of first image into the level
    used_width[0]=image_width[current_image];    //update used width of first level by adding width of image just packed

    for(int i=1;i<num_images;i++){  //for the remaining num_images-1 images
        current_image=images[i];        //choose the image with next highest height

        if(texture_width-used_width[current_lvl] < image_width[current_image] || lvl_height[current_lvl] < image_height[current_image]){ //if the current level is to short or has too little unused width remaining
            current_y+=lvl_height[current_lvl];    //update current_y by adding the height of the previous level
            current_lvl++;  //move to the next level
            lvl_height[current_lvl]=image_height[current_image];    //set the height of the new level based on the height of the current image
        }
            x_pos[current_image]=used_width[current_lvl];   //set the x position of the current image to be as far left as possible in the current level
            y_pos[current_image]=current_y;     //set the y position of the current image to be the y position of the current level
            used_width[current_lvl]+=image_width[current_image];     //update the amount of width already used in the current level by adding the width of the current image
    }
    current_y+=lvl_height[current_lvl];     //calculates the total height of the packing
    return current_y;   //returns the total height of the packing
}

int* sort(minheap h){     //takes as arguements a minheap h to be sorted
    while(!isEmpty(h))  pop(h);    //pops elements until there are none left
    return h->nodes+1;  //when the heap is empty, the array of elements in the heap will be in decreasing order. the 0th node is a sentinel. therefore, we return h->nodes[1:]
}

minheap buildheap(){
    minheap h=(minheap)malloc(sizeof(struct minheap_struct));   //allocates memory for the minheap structure
    h->capacity=num_images;  //sets the max capacity and size of the structure to be equal to the size of the structure
    h->size=num_images;
    h->nodes=(int*)malloc(sizeof(int)*(num_images+1));   //allocates memory for the nodes of the minheap
    h->nodes[0]=-1; //assigns the value -1 to the sentinel node at position 0
    h->nodes[1]=0;  //assigns the address of the 0th image to the root of the minheap, adding the first no
de to the minheap

    int j, minchild;    //j is an iterator used to travel down the tree while "percolating down". minchild
 stores the minimum child of node while percolating down
    for(int i=h->size;i>0;i--){
        j=i;    //start at the next empty position in the tree
        while(2*j < h->size){   //while the current node has children
            minchild=2*j;   //assume the left child has smallest height
            if(minchild+1<h->size && image_height[h->nodes[minchild]] > image_height[h->nodes[minchild+1]]
)     minchild++;   //if the height of the right child is smaller, update minchild
            if(image_height[i-
1] > image_height[h->nodes[minchild]]){     //if the height of the node to be inserted is less than the sm
allest of the heights of its children
                h->nodes[j]=h->nodes[minchild]; //percolate the child up
                j=minchild; //move down a level
            }
            else    break;  //the current node has smaller height than both of its children
        }
        h->nodes[j]=i-1;    //insert the new node at the position found
    }   // !!!NOTE!!! that the values that are stored in the min heap are not the heights of the images, b
ut rather the addresses of each image in the heights and widths arrays. The nodes are sorted however accor
ding to their heights.
    return h;   //return the built heap
}

int pop(minheap h){   //takes as arguements a heap h to be popped from

    int temp=h->nodes[h->size]; //saves the last item in the heap
    h->nodes[h->size]=h->nodes[1];  //moves the item at the top of the heap to the former last position in
 the array which will now be outside of the heap
    h->size--;  //decreases the heap size, excluding the last position in the array from the heap

    int i=1, minchild;  //i is an iterator used to travel down the heap while percolating down, minchild i
s used to store the child of the current node with smallest height
    //i is set to the top node in the heap
    while(i*2<h->size+1){   //while the current node has children
        minchild=i*2;   //assume the left child has smallest height
        if(minchild<h->size && image_height[h->nodes[minchild]] > image_height[h->nodes[minchild+1]])
minchild++;   //if the height of the right child is smaller, update minchild
        if(image_height[temp] > image_height[h->nodes[minchild]]){      //if the former last node in the a
rray has greater height than both children of the current node
            h->nodes[i]=h->nodes[minchild]; //percolate the child with minimum height up
            i=minchild; //move down a level
        }
        else break;     //the current child has no children
    }
    h->nodes[i]=temp;   //insert the former last node at the position found
    return h->nodes[h->size+1];     //return the former top node
}

int isEmpty(minheap h){     //takes a min heap as an arguement
    if(h->size==0)  return 1;   //empty
    else return 0;  //not empty
}
```

Declaration

We hereby declare that all the work done in this project titled "Project Five: Texture Packing" is of our independent effort as a group.

Duty Assignments

Programmer:

Tester:

Report Writer:

References

1. *Coffman, Jr. E. G., M. R. Garey, D. S. Johnson, and R. E. Tarjan. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms." SIAM Journal on Computing 9, no. 4 (1980): 808–26. https://doi.org/10.1137/0209062.*
2. *Garey, M. R., and D. S. Johnson. "Approximation Algorithms for Bin Packing Problems: A Survey." Analysis and Design of Algorithms in Combinatorial Optimization, 1981, 147–72. https://doi.org/10.1007/978-3-7091-2748-3_8.*
3. *Bansal, Nikhil, José R. Correa, Claire Kenyon, and Maxim Sviridenko. "Bin Packing in Multiple Dimensions: Inapproximability Results and Approximation Schemes." Mathematics of Operations Research 31, no. 1 (2006): 31–49. https://doi.org/10.1287/moor.1050.0168.*
4. *Hofri, Micha. "Two-Dimensional Packing: Expected Performance of Simple Level Algorithms." Information and Control 45, no. 1 (1980): 1–17. https://doi.org/10.1016/s0019-9958(80)90817-7.*
5. *"Survey for 2-D Packing Algorithms." ARC project. Accessed May 3, 2020. https://cgi.csc.liv.ac.uk/~epa/surveyhtml.html.*