Analysis of Algorithms

BLG 335E

Project 2 Report

İbrahim Karateke
150210705
karateke21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 14.12.2023

1. Implementation

In this part, we will go into the implementation details of our code step by step. Each part of the code and the logic behind those code snippets will be explained elaborately.

1.1. Handling Data

Since our data is provided in .csv files firstly we need to perform some adjustments in order to process the data. We started by including the required libraries. All libraries that are included have different purposes and those can be listed as:

- **iostream**: To be able to use basic print operation
- **vector**: To be able to handle data and to create a collection for heap operations.
- **fstream**: To be able to read the data from .csv files and to write the outputs to the desired files
- **sstream**: To be able to organize the read lines from files in particular splitting them
- **chrono**: To be able to measure how much time taken by the implemented functions

```cpp
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <chrono>
```

Our provided data is composed of some pairs which have the city names and their population. Therefore, we need to define our atomic datatype which keeps the information about each instance in this context. The data type is defined by following the code snippet.

```cpp
typedef std::pair<std::string, int> CityPair;
```

In .csv files each attribute in each row is split by semicolon (;). Hence, we need to have a **split** function that can distinguish each attribute whenever the program encounters with a semicolon during the reading of the .csv file. This function simply takes two parameters, namely the current line and the delimiter that splits the attributes. In our case, delimiter will be a semicolon. The code snippet of this function can be seen below.

```cpp
std::vector<std::string> split(const std::string &s, char delimiter) {
    std::vector<std::string> tokens;
    std::stringstream ss(s);
    std::string token;
    while (std::getline(ss, token, delimiter)) {
        tokens.push_back(token);
    }
    return tokens;
}
```

The **csvToVector** function is designed to extract data from a .csv file containing city-population pairs and populate a vector of CityPair objects.

The function starts by opening the specified .csv file using an input file stream. It checks if the file is opened successfully. If there's an issue opening the file, it outputs an error message and stops further execution. Next, it checks for a BOM (Byte Order Mark) in the file by reading the first three bytes. If a BOM is detected (specific UTF-8 sequence), it adjusts the file reading position to skip these bytes. If no BOM is found, it resets the file position to the start. Then, the function enters a loop to read each line from the file. For each line, it splits the content into a vector of strings (row) based on the semicolon delimiter (;). It verifies if the row contains exactly two elements, each row represents a city and its population. When a city-population pair is found, it extracts the city name and population. It converts the population string into an integer and creates a CityPair object with the city name and population. This CityPair object is appended to the data vector. Finally, after reading all the data, the function closes the file stream. The code snippet of this function can be seen below.

```cpp
void csvToVector(std::string filename,std::vector<CityPair> &data){
    std::ifstream file(filename);

    if (!file.is_open()) {
        std::cerr << "Error opening file!" << std::endl;
        return;
    }

    char bom[3] = {0};
    file.read(bom, 3);
    if (bom[0] == '\xEF' && bom[1] == '\xBB' && bom[2] == '\xBF') {
        file.seekg(3);
    } else {
        file.seekg(0);
    }

    std::string line;
    while (std::getline(file, line)) {
        std::vector<std::string> row = split(line, ';');
        if (row.size() == 2) {
            std::string city = row[0];
            int population = std::stoi(row[1]);
            CityPair cityData = std::make_pair(city, population);
            data.push_back(cityData);
        }
    }

    file.close();
}
```

In order to keep our processed data, we need to have a function called **output**. This function simply takes two parameters which are the processed data that is going to be kept in .csv file and the name of the output file. If there is a same-named file in the folder of our program it removes this file and creates a new output file for the current running of the program. It simply traverses the whole collection starting from the beginning of it. Then, it combines city names and their population by adding a semicolon between them. In the end, the format of the data in the output file will be exactly the same as the initial format of the unprocessed data. The code snippet of this function can be seen below.

```cpp
void output(std::vector<CityPair>& data, std::string outputFileName){
    remove(outputFileName.c_str());
    std::ofstream myfile;
    myfile.open(outputFileName, std::ios_base::app);
    for (int i = 0; i < data.size(); ++i) {
        myfile << data[i].first << ";" << data[i].second;
        if (i != data.size() - 1) {
            myfile << "\n";
        }
    }
    myfile.close();
}
```

1.2. Building Max Heap

In this part, we will go into detail about how the heap data structure is created. Throughout this project, heap will be our main data structure and we perform all operations in this data structure. In order to build a heap from any kind of collection, whether sorted or unsorted, we need to have a **build_max_heap** function. This function will take two parameters namely, the collection itself and the degree of a heap. Since later we need to construct d-ary heaps, we need to generalize this function so that the created heap might be binary, ternary, and so on (i.e. d-ary). Building max heap starts from the last non-leaf node in the heap. Once we find the last non-leaf node, it needs to check whether its children nodes are greater than it or not. If any of its children is greater than this node it swaps them. Since we started from the last non-leaf node any decrementation that will occur during the traversing will correspond to another non-leaf node and the same process will be performed on other nodes. The code snippet of this function can be seen on the next page.

```cpp
void build_max_heap(std::vector<CityPair>& data, int d) {
    int n = data.size();
    int lastNonLeaf;
    if (n % d==0){
        lastNonLeaf=n/d -1;
    }
    else{
        lastNonLeaf=n/d;
    }

    for (int i = lastNonLeaf; i >= 0; i--)
        max_heapify(data, n, i, d);
}
```

As it can be inferred from the code, this function only determines the very last non-leaf nodes. Comparisons between parent and its children are performed by **max_heapify** function. This function takes four parameters namely, the collection, the size of the collection, the index of the parent node, and the degree of the heap. If there exists a child which is greater than the parent it swaps them and the child becomes parent and vice versa. Notice that there might exist a child which is greater than other children. Since that child became the parent the comparisons will continue until the largest child becomes the parent (see the recursion). This process will continue until we reach the root node, which is the largest node at the end we will have a structure in which all parents are larger than their children. Hence we got a max heap structure. The code snippet of this function can be seen below.

```cpp
void max_heapify(std::vector<CityPair>& data, int n, int i, int d) {
    int largest = i;
    int firstChild = d * i + 1;

    for (int j = 0; j < d; ++j) {
        int child = firstChild + j;
        if (child < n && data[child].second > data[largest].second)
            largest = child;
    }

    if (largest != i) {
        swap(data[i], data[largest]);
        max_heapify(data, n, largest, d);
    }
}
```

Those two functions perform the operation with the help of **swap** function. That function as its name implies swaps those two values between them. The code snippet of this function can be seen below.

```cpp
void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

1.3. Heapsort

In this part, we will go into detail about how heapsort is performed. Notice that we have dealt with the problem of building max heap and after heapification is done the maximum value will be placed at index 0 in the vector. In other words, it becomes the root. This function takes two parameters, namely the collection of the data and the degree of the heap. In **heapsort** function, we started building a max heap considering the case the input is not given as a heap. Afterwards, we started a loop that traverses the whole heap starting from the very last element (i.e. child). This swaps that element with the root which is the maximum element. Notice that the largest element is moved to the last index; hence sorting begins. Since this loop decrements in every iteration, when we call "max_heapify" function, heapification is going to be done on only the remaining element. The largest element that was positioned previously is not going to be touched. In a nutshell, with the help of "max_heapfiy" function root (first index) will be the largest element and it is going to be swapped with the last element in the context of the current iteration until we get a sorted collection. The code snippet of this function can be seen below.

```cpp
void heapsort(std::vector<CityPair>& data, int d) {
    int n = data.size();

    build_max_heap(data, d);

    for (int i = n - 1; i > 0; i--) {
        swap(data[0], data[i]);
        max_heapify(data, i, 0, d);
    }
}
```

1.4. Priority Queue Operations

In this part, we will go into detail about how priority queue operations are performed in heaps. At this stage, all the operations are assumed to be performed on binary heaps.

1.4.1. Heap Increase Key

The function named **heap_increase_key** basically increases the value of an element in the desired index. The function takes three parameters namely, the collection of the data, the index value of the element that is wanted to increase and the new key value. Considering the case data is not given as a heap, we started building a binary max heap. This is the reason why the "build_max_heap" is called with the value of 2. Then we check whether the new key is greater than the existing key or not. If it is not greater, no operation is done. If it is greater, the new value is replaced. However, this increasing operation might create a collection that violates the binary max heap properties. Hence, the replaced node is going to be compared with its parent and if it is greater than the parent it will swapped with the parent. This process will continue until the replaced node is placed in a correct position. In other words, the replaced node's parent will be greater than it. Thus, in the end, we will have a binary heap that does not violate the maximum binary heap properties with the increased key. The code snippet of this function can be seen below.

```cpp
void heap_increase_key(std::vector<CityPair>& data, int i, int key) {
    build_max_heap(data,2);
    if (key < data[i].second) {
        std::cout << "New population is smaller than the current population. NO
        ↪ INCREASE KEY OPERATION IS PERFORMED." << std::endl;
        return;
    }

    data[i].second = key;

    while (i > 0 && data[(i - 1) / 2].second < data[i].second) {
        swap(data[i], data[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}
```

1.4.2. Max Heap Insert

This function named **max_heap_insert** performs the insertion operation to the heap. This function takes two parameters namely, the collection of the data and the pair (city name and population) which is desired to be inserted into the heap. This function is quite similar to the "heap_increase_key" function. The only difference we do not change any value of a node but we add a new node into the heap. Considering the case data is not given as a heap, we started building a binary max heap. This is the reason why the "build_max_heap" is called with the value of 2. The new pair (node) is pushed into the collection; therefore, the new pair is placed in the last index. Just like in the previous function, we need to check whether the added pair creates a violation in terms of heap properties or not. Hence, the new node is going to be compared with its parent and if it is greater than the parent it will swapped with the parent. This process will continue until the new node is placed in a correct position. In other words, the new node's parent will be greater than it. Thus, in the end, we will have a binary heap that does not violate the maximum binary heap properties with the new node. The code snippet of this function can be seen below.

```
void max_heap_insert(std::vector<CityPair>& data,std::pair<std::string, int>
↪   &CityPair) {
    build_max_heap(data,2);
    data.push_back(CityPair);
    int index=data.size()-1;

    while (index > 0 && data[(index - 1) / 2].second < data[index].second) {
        swap(data[index], data[(index - 1) / 2]);
        index = (index - 1) / 2;
    }
}
```

1.4.3. Heap Maximum

This function named **heap_maximum** is going to return a pair which has the largest value. In other words, it returns the most crowded city with the city name and the population information. The function basically takes only one parameter which is the collection of the data. Considering the case data is not given as a heap, we started building a binary max heap. This is the reason why the "build_max_heap" is called with the value of 2. Since we construct a maximum binary heap, it is clear that the largest value is placed in 0^{th} index (i.e. root). Hence this value is returned. The code snippet of this function can be seen on the next page.

```
std::pair <std::string,int> heap_maximum(std::vector<CityPair>& data){
    build_max_heap(data,2);
    return data[0];
}
```

1.4.4. Heap Extract Max

This function named **heap_extract_max** is going to return a pair which has the largest value. In other words, it returns the most crowded city with the city name and the population information. The function basically takes only one parameter which is the collection of the data. Considering the case data is not given as a heap, we started building a binary max heap. This is the reason why the "build_max_heap" is called with the value of 2. Since we construct a maximum binary heap, it is clear that the largest value is placed in 0th index (i.e. root). Hence, this value is assigned to a variable that is going to be returned. Then, the root is swapped with the last element of this heap. Now we have a relatively very small value in the root compared to its children. Starting from the root "max_heapify" function is called in order to maintain heap structure. Once this process is done, the previously assigned value is returned as the maximum pair. The code snippet of this function can be seen below.

```
std::pair <std::string,int> heap_extract_max(std::vector<CityPair>& data){
    build_max_heap(data,2);
    std::pair <std::string,int> ret=data[0];
    swap(data[0], data[data.size() - 1]);
    data.pop_back();
    max_heapify(data, data.size(),0,2);
    return ret;
}
```

1.5. Implementation of d-ary Heap Operations

In this part, we will go into detail about priority queue operations in d-ary heaps. Notice that since we implemented our "build_max_heap" function in a way that can be generalized to any number of degrees those functions will follow the almost same fashion as the previous functions do. d-ary heaps can be defined as a generalization of the binary heap in which the nodes have d children instead of 2.

1.5.1. Height Calculation

The function called **dary_calculate_height** calculates the height of our d-ary heap. In order to perform this operation we need to have two pieces of information which are the number of elements in the collection and the number of ’d’ which corresponds to the degree of the heap. Hence, the function takes those two parameters as the inputs and returns an integer which is the height of the d-ary heap after the calculation is done. In order to calculate the height the collection does not have to be a heap, only those two pieces of information are enough to determine the height. It initializes variables to track the height and the number of nodes at the current level. The algorithm iterates through the tree levels, decrementing the node count by the number of nodes at each level, updating the count of nodes at the subsequent level by multiplying the arity and incrementing the height. Finally, it returns the height of the tree in terms of edges, subtracting 1 to account for counting edges instead of nodes in the tree height. The code snippet of this function can be seen below.

```c
int dary_calculate_height (int n, int d) {
    int height = 0;
    int nodes_at_current_level = 1;

    while (n > 0) {
        n =n- nodes_at_current_level;
        nodes_at_current_level = nodes_at_current_level * d;
        height++;
    }

    return height - 1;
}
```

1.5.2. d-ary Heap Increase Key

The function called **dary_increase_key** function works in almost the same way as the "heap_increase_key" does. However, there are a few differences. For example, this function takes an additional parameter which is "d". Because our heap is not restricted to a binary heap anymore. Similarly, this parameter is sent also to the "build_max_heap" function in order to build d-ary heap. In the "heap_increase_key" function, during the iteration, we were dividing our index values by 2. Nevertheless, in this function, we need to divide those values by d since our heap is a d-ary heap. Other operations are done on the same logic with the function that is for the binary heap. The code snippet of this function can be seen on the next page.

```cpp
void dary_increase_key(std::vector<CityPair>& data, int i, int key, int d) {
    build_max_heap(data,d);
    if (key < data[i].second) {
        std::cout << "New population is smaller than the current population. NO
        ↪  INCREASE KEY OPERATION IS PERFORMED." << std::endl;
        return;
    }

    data[i].second = key;

    while (i > 0 && data[(i - 1) / d].second < data[i].second) {
        swap(data[i], data[(i - 1) / d]);
        i = (i - 1) / d;
    }
}
```

1.5.3. d-ary Heap Insert

The function called **dary_insert_element** function works in almost the same way as the "heap_max_insert" does. However, there are a few differences. For example, this function takes an additional parameter which is "d". Because our heap is not restricted to a binary heap anymore. Similarly, this parameter is sent also to the "build_max_heap" function in order to build d-ary heap. In the "heap_increase_key" function, during the iteration, we were dividing our index values by 2. Nevertheless, in this function, we need to divide those values by d since our heap is a d-ary heap. Other operations are done on the same logic with the function that is for the binary heap. The code snippet of this function can be seen below.

```cpp
void dary_insert_element(std::vector<CityPair>& data,std::pair<std::string, int>
↪  &CityPair, int d) {
    build_max_heap(data,d);
    data.push_back(CityPair);
    int index=data.size()-1;

    while (index > 0 && data[(index - 1) / d].second < data[index].second) {
        swap(data[index], data[(index - 1) / d]);
        index = (index - 1) / d;
    }

}
```

1.5.4. d-ary Extract Max

The function called **dary_extract_max** function works in almost the same way as the "heap_extract_max" does. However, there are a few differences. Just like the previous two functions, it also takes an additional parameter "d". Because our heap is not restricted to a binary heap anymore. Whether binary or d-ary heap, since the maximum value is placed on 0^{th} index, that value is going to be returned as the maximum pair, and the same pair is going to be removed from the collection. The code snippet of this function can be seen below.

```
std::pair <std::string,int> dary_extract_max(std::vector<CityPair>& data,int d){
    build_max_heap(data,d);
    std::pair <std::string,int> ret=data[0];
    swap(data[0], data[data.size() - 1]);
    data.pop_back();
    max_heapify(data, data.size(),0,d);
    return ret;
}
```

1.5.5. Main Function

In the project document, we are asked to pass arguments to the implemented function as parameters. Therefore, we started to assign our first fixed three arguments into variables. According to the both project document and the main function **argv[1]**, **argv[2]** and **argv[3]** correspond to the initial data set (might be a heap, a sorted collection or an unsorted collection), function name that is desired to be run and output file name, respectively. Furthermore, those three arguments are assigned to the related variables. Notice that the order of those three arguments is assumed fixed and the program does not run without those three arguments. Moreover, the initial data set is moved to our previously created pair vector. Hence it is ready to perform operations on it.

There might exist some optional arguments like ’d’, ’i’ and ’k’ which are the number of children of non-leaf nodes, the index of a node and the key value to update an element, respectively. They are optional and the interpretation of given values depends on the name of the function. When providing any of these three parameters as an argument, their values are concatenated to the letter, e.g. ’d5’, ’i125’ or ’k43985’. In order to deal with this problem, we need to check those arguments with if blocks to make sure the fact that the correct argument is assigned to the correct variable according to the first character of the argument. Notice that for further operations we only consider the numeric part of the argument. Thus, before the assignment, we take the substring of those and convert those to integers.

Since in the project document, it is emphasized that **"Note that indices begin with 1 in the context of the heap. Therefore if 1 is given as index, it corresponds to the root of the heap"** we need to subtract 1 before assigning that value to 'i'. The reason for this operation is that our data structure works with a vector hence the root of the heap that is entered as 1 by the user corresponds to the 0^{th} index in our heap. Furthermore, this fashion is followed for all other nodes that exist on the heap. A collapsed version of this function can be seen below. We will go over each condition one by one because each function has different behaviors in its context.

```cpp
int main(int argc, char* argv[]) {
    std::string fileName= argv[1];
    std::string functionName=argv[2];
    std::string outFileName=argv[3];
    int d=2;
    int i,k;
    int insertArgument;
    std::vector<CityPair> data;
    csvToVector(fileName,data);
    for(int j=4; j<argc;j++){
        std::string currentArgument=argv[j];
        if(currentArgument[0]=='d'){
            d=stoi(currentArgument.substr(1));
        }
        if(currentArgument[0]=='i'){
            i=stoi(currentArgument.substr(1))-1;
        }
        if(currentArgument[0]=='k' && functionName!="max_heap_insert" &&
         ↪  functionName!="dary_insert_element"){
            k=stoi(currentArgument.substr(1));
        }
        if(currentArgument[0]=='k' && (functionName=="max_heap_insert" ||
         ↪  functionName=="dary_insert_element")){
            insertArgument=j;
        }
    }
    if(functionName=="build_max_heap"){...}
    else if (functionName=="heapsort"){...}
    else if(functionName=="max_heap_insert"){...}
    else if(functionName=="heap_increase_key"){...}
    else if (functionName=="heap_extract_max"){...}
    else if (functionName=="heap_maximum"){...}
    else if(functionName=="dary_calculate_height"){...}
    else if(functionName=="dary_extract_max"){...}
    else if(functionName=="dary_insert_element"){...}
    else if(functionName=="dary_increase_key"){...}
    return 0;
}
```

1. **If the function is "build_max_heap":**

If **argv[2]** is entered as **build_max_heap** by the user, then we only call the build_max_heap function. However, there may or may not be a 'd' value specified by the user. Therefore, if something is specified the function will called with that value and it builds a d-ary heap. Otherwise, since we assigned 2 to the 'd' by default it will build a binary heap. Afterwards, we call the **output** function to save the built max heap into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
if(functionName=="build_max_heap"){
        build_max_heap(data,d);
        output(data,outFileName);
    }
```

2. **If the function is "heapsort":**

If **argv[2]** is entered as **heapsort** by the user, then we call the heapsort function. However, there may or may not be a 'd' value specified by the user. Therefore, if something is specified the function will called with that value and it builds a d-ary heap. Otherwise, since we assigned 2 to the 'd' by default it will build a binary heap. Notice that since heapsort will be performed after building a dimension-specified heap within itself we need to consider this case and call the function accordingly. There are some parts that are taken into comment, those parts are only used for measuring the time with different inputs. Afterwards, we call the **output** function to save the sorted collection into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
    else if (functionName=="heapsort"){
        //const auto start{std::chrono::steady_clock::now()};
        heapsort(data,d);
        //const auto end{std::chrono::steady_clock::now()};
        //const std::chrono::duration<long long,std::nano> elapsed_seconds{end -
        ↪  start};
        //std::cout<<"Time taken by Heapsort via constructing "<<d<<"-ary heap:
        ↪  "<<elapsed_seconds.count()<<" ns."<<std::endl;
        output(data,outFileName);
    }
```

3. **If the function is "max_heap_insert":**

If **argv[2]** is entered as **max_heap_insert** by the user, then we call the max_heap_insert function. This function is going to be performed only in binary heap. Notice that, we extracted the 'k' value in the beginning of our main function. However, for the insert operation, the nature of 'k' is different. In this case, 'k' will be provided as k_<cityname>_<population>. By considering this possibility, we previously assigned the index of 'k' argument into a variable called **insertArgumet**. Hence, we know the exact location of this argument in terms of what the user entered. At this point, we again process 'k' value and split those two attributes namely city name and population with respect to the underscores. After dealing with assignments, we combine those two attributes into a pair and send it to our heap. Finally, we call the **output** function to save the heap with the inserted value into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```cpp
    else if(functionName=="max_heap_insert"){
        std::string currentArgument=argv[insertArgument];
        currentArgument=currentArgument.substr(1);
            // Find the position of the underscores
        int underscore1 = currentArgument.find('_');
        int underscore2 = currentArgument.find('_', underscore1 + 1);

        // Extract the city name and the number using substr
        std::string city = currentArgument.substr(underscore1 + 1, underscore2 -
        ↪  underscore1 - 1);
        int population=stoi(currentArgument.substr(underscore2 + 1));

        CityPair insertedPair = std::make_pair(city, population);
        max_heap_insert(data,insertedPair);
        output(data,outFileName);
    }
```

4. **If the function is "heap_increase_key":**

If **argv[2]** is entered as **heap_increase_key** by the user, then we call the heap_increase_key function. This function is going to be performed only in binary heap. In this case, 'i' and 'k' must be specified by the user. Note that 'i' corresponds to the index of the heap that is wanted to be changed and 'k' is the increased value of that index (i.e. value of new population of this city). Note that if the increase operation is not valid, that is, if the new key value is not greater than the previous one, this is not going to be done. However, since we build a heap by calling the heap_increase_key function just in case, at the end the data will be heapified. Finally, we call the **output** function to save the heap with the increased key value or just heapified collection into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen on the next page.

```
    else if(functionName=="heap_increase_key"){
        heap_increase_key(data,i,k);
        output(data,outFileName);
    }
```

5. **If the function is "heap_extract_max":**

If **argv[2]** is entered as **heap_extract_max** by the user, then we call the heap_extract_max function. This function is going to be performed only in binary heap. Notice that this function returns a pair which has the maximum population with the corresponding city name. However, this function also removes that returned and maintains the heap structure. Since our **output** function takes input as a vector of pair we push this pair into an empty vector. Finally we send this vector as an input to the output function to save this into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
    else if (functionName=="heap_extract_max"){
        CityPair maximumValue=heap_extract_max(data);
        std::vector<CityPair> onlyMaximumVector;
        onlyMaximumVector.push_back(maximumValue);
        output(onlyMaximumVector,outFileName);
    }
```

6. **If the function is "heap_maxium":**

If **argv[2]** is entered as **heap_maximum** by the user, then we call the heap_maximum function. With this function, we perform exactly the same operation that we did in the previous part. The only difference is that the maximum pair is not removed from the heap. The code snippet of this part can be seen below.

```
    else if (functionName=="heap_maximum"){
        CityPair maximumValue=heap_maximum(data);
        std::vector<CityPair> onlyMaximumVector;
        onlyMaximumVector.push_back(maximumValue);
        output(onlyMaximumVector,outFileName);
    }
```

7. **If the function is "dary_calculate_height":**

If **argv[2]** is entered as **dary_calculate_height** by the user, then we call the dary_calculate_height function. This function only returns a single integer which is the height of d-ary heap. Since our **output** function does not match with the format of this output, we simply indicate the result in both console output and .csv file output. Hence the result will be saved on a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
    else if(functionName=="dary_calculate_height"){
        remove(outFileName.c_str());
        std::ofstream myfile;
        myfile.open(outFileName, std::ios_base::app);
        int height =dary_calculate_height(data.size(),d);
        myfile<<"There are "<<data.size()<<" elements in this "<<d<<"-ary heap. Its
        ↪  height is "<<height<<".";
        myfile.close();
        std::cout<<"There are "<<data.size()<<" elements in this "<<d<<"-ary heap.
        ↪  Its height is "<<height<<"."<<std::endl;
    }
```

8. **If the function is "dary_extract_max":**

If **argv[2]** is entered as **dary_extract_max** by the user, then we call the dary_extract_max function. What is done in the part is almost the same as in **heap_extract_max**. The only difference is that we consider ’d’ value since we are creating d-ary heaps. Notice that although ’d’ is not specified by the user, if this function is called it will be performed considering d is 2. Because d is specified as 2 by default in the main function. Either a binary or d-ary since the maximum value depends on the data itself the returned value will be the same. Finally, after adjusting the formatting just like we did in heap_extract_max the result will be saved on a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
    else if(functionName=="dary_extract_max"){
        CityPair maximumValue=dary_extract_max(data,d);
        std::vector<CityPair> onlyMaximumVector;
        onlyMaximumVector.push_back(maximumValue);
        output(onlyMaximumVector,outFileName);
    }
```

9. **If the function is "dary_insert_element":**

If **argv[2]** is entered as **dary_insert_element** by the user, then we call the dary_insert_element function. What is done in the part is almost the same as in **max_heap_insert**. The only difference is that we consider 'd' value since we are creating d-ary heaps. Notice that although 'd' is not specified by the user, if this function is called it will be performed considering d is 2. Because d is specified as 2 by default in the main function. Finally, we call the **output** function to save the heap with the inserted value into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
    else if(functionName=="dary_insert_element"){
        std::string currentArgument=argv[insertArgument];
        currentArgument=currentArgument.substr(1);
            // Find the position of the underscores
        int underscore1 = currentArgument.find('_');
        int underscore2 = currentArgument.find('_', underscore1 + 1);

        // Extract the city name and the number using substr
        std::string city = currentArgument.substr(underscore1 + 1, underscore2 -
        ↪  underscore1 - 1);
        int population=stoi(currentArgument.substr(underscore2 + 1));
        CityPair insertedPair = std::make_pair(city, population);
        dary_insert_element(data,insertedPair,d);
        output(data,outFileName);
    }
```

10. **If the function is "dary_increase_key":**

If **argv[2]** is entered as **dary_increase_key** by the user, then we call the dary_increase_key function. What is done in the part is almost the same as in **heap_increase_key**. The only difference is that we consider 'd' value since we are creating d-ary heaps. Notice that although 'd' is not specified by the user, if this function is called it will be performed considering d is 2. Because d is specified as 2 by default in the main function. Finally, we call the **output** function to save the heap with the inserted value into a .csv file with the desired name according to **argv[3]** or **outFileName**. The code snippet of this part can be seen below.

```
    else if(functionName=="dary_increase_key"){
        dary_increase_key(data,i,k,d);
        output(data,outFileName);
    }
```

2. Compiling and Running

We compiled the program with the following shell command.

```
g++ -o Heapsort Heapsort.cpp
```

Once the compilation is done, we run the program with the following shell command.

```
./Heapsort dummyData.csv heap_increase_key out.csv i1 k125
```

Notice that the program might be run with different arguments depending on the function name or argv[2] that is entered. For example, in the above command, we run the heap_increase_key function and we want to increase the root's population value to 125. The function name and corresponding arguments regarding that function may vary.

Furthermore, this is important that in this program whenever the program is run, if the output file name is specified with the same name just like in the previous running, the older output file is deleted and a new file with the same name is created. Therefore, if the output is wanted to be used in the next run as an input the pattern which is below should be followed.

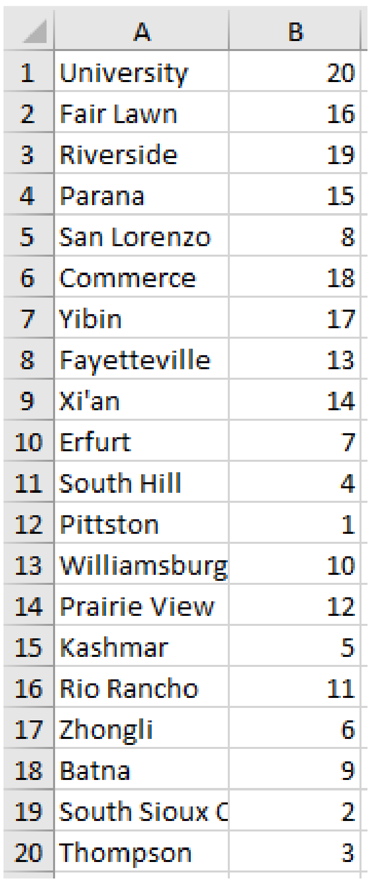
```
./Heapsort dummyData.csv build_max_heap out.csv
./Heapsort out.csv max_heap_insert out2.csv i1 k125
```

Thus, in **out.csv** can be seen the max binary heap of dummyData.csv and in **out2.csv** can be seen the root's population value increased to 125.

3. Results

In this part, we are going to examine some outputs of our code with the provided arguments.

```
./Heapsort dummyData.csv build_max_heap out.csv
```



	A	B
1	University	20
2	Fair Lawn	16
3	Riverside	19
4	Parana	15
5	San Lorenzo	8
6	Commerce	18
7	Yibin	17
8	Fayetteville	13
9	Xi'an	14
10	Erfurt	7
11	South Hill	4
12	Pittston	1
13	Williamsburg	10
14	Prairie View	12
15	Kashmar	5
16	Rio Rancho	11
17	Zhongli	6
18	Batna	9
19	South Sioux C	2
20	Thompson	3

Figure 3.1: Content of out.csv after building binary heap

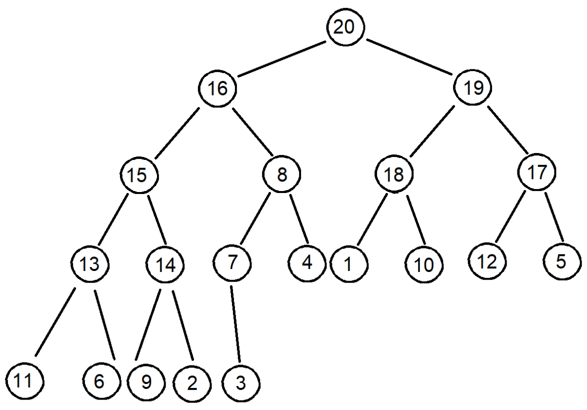
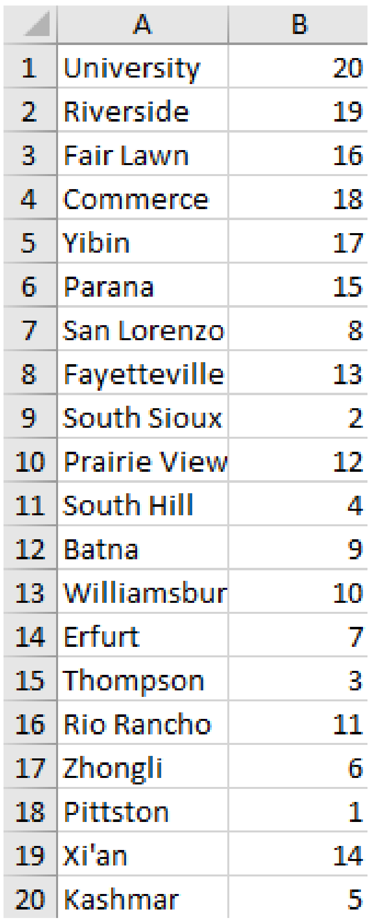


Figure 3.2: Binary heap

```
./Heapsort dummyData.csv build_max_heap out.csv d3
```



	A	B
1	University	20
2	Riverside	19
3	Fair Lawn	16
4	Commerce	18
5	Yibin	17
6	Parana	15
7	San Lorenzo	8
8	Fayetteville	13
9	South Sioux	2
10	Prairie View	12
11	South Hill	4
12	Batna	9
13	Williamsbur	10
14	Erfurt	7
15	Thompson	3
16	Rio Rancho	11
17	Zhongli	6
18	Pittston	1
19	Xi'an	14
20	Kashmar	5

Figure 3.3: Content of out.csv after building ternary heap

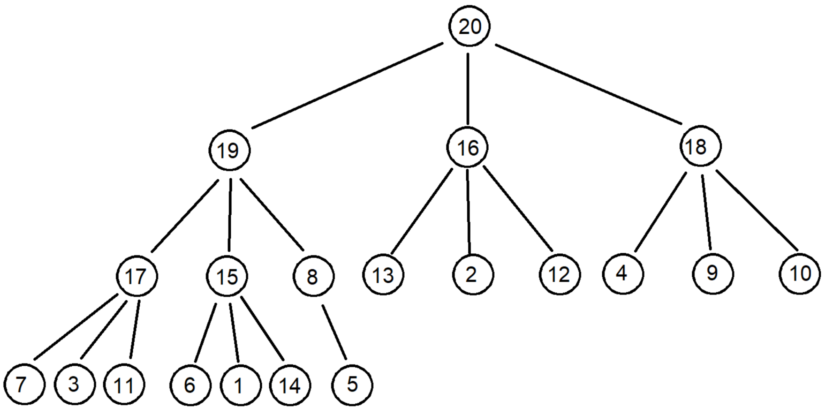


Figure 3.4: Ternary heap "d=3"

4. Discussion

In the **dary_extract_max** function, after swapping the last element with the root, the subsequent process of **max_heapify** involves comparing the node with its children to maintain the heap property. This operation, crucial to restoring the max-heap structure, proceeds in a single branch, specifically moving towards the leaf level where the heap property is eventually restored. This process iterates through the height of the tree, adjusting elements as needed to ensure the maximum value remains at the root. Thus, considering a d-ary heap, the time complexity for this process is indeed $O(\log_d n)$ where n is the number of elements in the collection and d is the degree of the heap.

In the **dary_increase_key** function, the approach involves updating the key at the specified index and then traversing upwards through the tree towards the root to restore the max-heap property if needed. This method differs from our **max_heapify** function. Instead of calling max_heapify function, it iterates upward, swapping the element with its parent as long as the heap property is violated, ensuring that the new key value is appropriately placed in the heap. This process continues until either the root is reached or the heap property is satisfied. The time complexity for this operation remains $O(\log_d n)$, where n denotes the number of nodes in the heap and d signifies the degree of the heap.

Similarly, the **dary_insert_element** function operates by appending the new element to the end of the heap and then traversing upwards through the tree to maintain the max-heap property. It involves swapping the newly inserted element with its parent until the heap property is satisfied. Similar to dary_increase_key, this upward traversal avoids calling max_heapify and ensures that the new element is correctly positioned in the heap. The time complexity for this insertion process also remains $O(\log_d n)$, considering n as the number of nodes in the heap and d as the degree of the heap.

5. Comperative Analysis

In the previous project, with the quicksort algorithm, we found the running time of the quicksort with different pivoting strategies in nanoseconds.

	Population1	Population2	Population3	Population4
Last Element	297,263,600	4,439,495,200	2,562,865,600	8,624,800
Random Element	7,906,400	6,914,800	7,196,900	8,199,700
Median of 3	6,773,000	6,658,00	7,325,700	8,043,100

Table 5.1: Comparison of different pivoting strategies on input data.

In this project, the running time of the heapsort algorithm over the same dataset in nanoseconds can be seen below.

	Population1	Population2	Population3	Population4
d=2	13,091,300	13,507,800	13,519,400	15,975,000
d=5	8,232,200	8,441,900	8,717,200	10,637,700
d=10	9,678,100	9,558,500	9,125,700	10,637,700
d=50	21,671,800	21,623,600	21,590,700	22,663,900
d=100	32,014,200	32,889,300	28,201,300	33,703,500

Table 5.2: Comparison of different d-ary heapsorts on input data.

Quicksort, on average, quicksort exhibits better performance than heapsort. quicksort's average time complexity is $O(n \log n)$, and in many cases, it performs exceptionally well due to its efficient partitioning and recursive nature. However, in the worst-case scenario, quicksort can degrade to $O(n^2)$ if the pivot selection consistently results in unbalanced partitions, which can happen with already sorted or nearly sorted data.

Heapsort, with its guaranteed $O(n \log n)$ time complexity in all cases, might outperform quicksort in scenarios where worst-case behavior matters. It maintains a consistent time complexity regardless of the initial order of the elements. However, heapsort tends to have a larger constant factor hidden in its Big O notation, which can make it slightly slower than quicksort for smaller input sizes or average cases.

In summary, quicksort tends to perform better on average, but its worst-case behavior might be a concern. Heapsort, while consistent, might be slightly slower due to its inherent overhead but guarantees $O(n \log n)$ performance in all scenarios. It is clear that the choice between these two algorithms often depends on the specific use case and the characteristics of the data being sorted.

Heapsort and quicksort have distinct strengths and weaknesses, making each preferable in different scenarios.

- **Quicksort Strengths:**

Efficiency: Quicksort typically outperforms heapsort in average cases due to its $O(n \log n)$ average time complexity and constant factors that are usually smaller.
Adaptability: It's adaptable to optimizations like random pivot selection or median-of-three pivot selection, reducing the likelihood of worst-case behavior.
Space Complexity: Quicksort has a smaller space overhead compared to heapsort, especially in-place partitioning schemes.

- **Quicksort Weaknesses:**

Worst-case Scenario: Its worst-case time complexity of $O(n^2)$ can occur with specific datasets (already sorted or nearly sorted), making it less preferable for scenarios where worst-case performance matters significantly.

- **Heapsort Strengths:**

Consistency: Heapsort guarantees $O(n \log n)$ time complexity in all cases, making it more suitable for scenarios where worst-case performance is crucial.
Stability: It is a stable sorting algorithm, meaning it maintains the relative order of equal elements, unlike quicksort. Worst-case Scenario: Heapsort's worst-case performance matches its average-case performance, ensuring consistent behavior across various datasets.

- **Heapsort Weaknesses:**

Constant Factor Overhead: Heapsort generally has a larger constant factor in its time complexity, making it slightly slower in practical scenarios compared to quicksort.
Space Complexity: It requires additional space to store the heap structure, making it less memory-efficient, especially for large datasets.

- **Scenarios:**

Quicksort: Preferred for average cases or when average-case performance matters more than the worst-case scenario. It's often used in general-purpose sorting libraries due to its speed and adaptability.
Heapsort: Better suited when guaranteed worst-case performance is critical, especially in real-time systems or safety-critical applications where consistent performance is prioritized over average-case efficiency. Also, its stability might be preferred in scenarios where maintaining the order of equal elements is crucial. Choosing between heapsort and quicksort often depends on the nature of the data, the importance of worst-case behavior, memory constraints, and the trade-offs between average-case efficiency and worst-case guarantees.