# Introduction to PR Quadtrees

Introduction

A quadtree is a data structure that stores d-dimensional points and enables efficient search for the stored points. We will only consider the case $d = 2$. One particular quadtree which can be used to store 2-dimensional points is the *point-region* quadtree, simply referred as a *PR quadtree*. A binary tree can be defined as a finite set of nodes that are either empty or have a root and two binary trees T_l and T_r (the left and right subtree). A quadtree has a similar recursive definition: instead of two subtrees we have four subtrees, hence the name quad. This means that each node either has four children or is a leaf node. The four leaf nodes are often referred to as NW, NE, SW, SE (see the figure below).
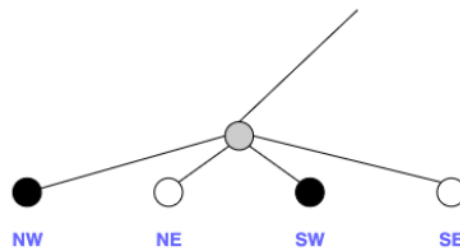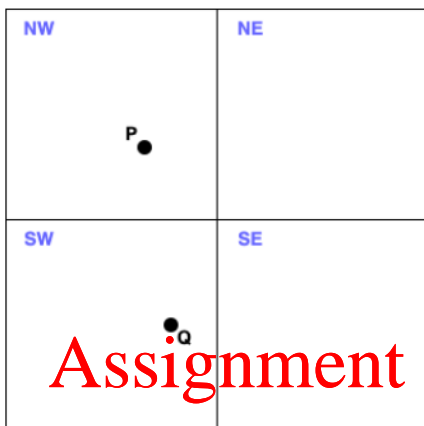
Building a PR Quadtree (Inserting Datapoints)

In a PR quadtree, each node represents a rectangle that covers part of the area that we wish to index. The root node covers the entire area. A PR quadtree is built recursively: we first start with an initial rectangle that should contain all the points we wish to store (how could you find a rectangle for n 2-dimensional points so that it contains all given datapoints?). To then construct a PR quadtree of a set of n points, we insert the points one by one and recursively subdivide a rectangle into four equally sized sub-rectangles whenever a rectangle would contain more than a single point. After the insertion process is complete, every leaf node is either coloured in *black* (contains a single datapoint) or *white* (indicating that the node is empty), and internal nodes are coloured in *grey* (see the figure above).

Alternatively, you could also use the following strategy: we insert all points into the rectangle and recursively partition each rectangle into 4 quadrants (sub-rectangles) until each rectangle contains at most one point.

The construction of a PR quadtree also leads to one important difference between them and binary trees: a binary tree depends on the order of value insertion whereas a PR quadtree is independent of the order of datapoint insertion. The reason is that

the nodes are independent from the inserted datapoints (the rectangles are determined by the initial rectangle of the root node).

Search

To enable efficient search, the rule is that *every leaf node in a PR quadtree either contains a single point or no point at all*. *Every node in a PR quadtree is either a leaf node or an internal node*, i.e., has four children representing a subdivision of the current rectangle into four equally sized quadrants. The region of an internal node always contains more than one point.

If we search for a single datapoint using its $2D$coordinates, we first check if the point lies in the quadtree. If it does, we recursively compute on each level the sub-rectangle that contains the coordinates of the datapoint to be searched until we reach a leaf node. The leaf node either contains the datapoint or it does not.

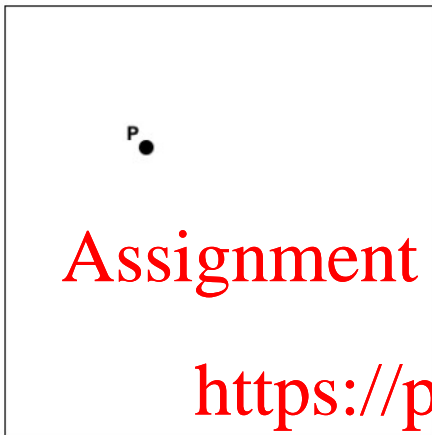我认为问题有可能出在这里 point 的存放问题

Assignment Project Exam Help

Example

https://powcoder.com

Here is a running example where we start with an empty PR quadtree, i.e., the we have a single white node, indicating we one leaf that is empty:

Add WeChat powcoder
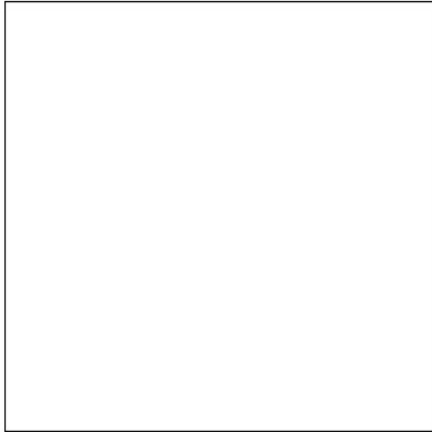
We insert the first point P. We can simply colour the root node as black as we have only a single point.
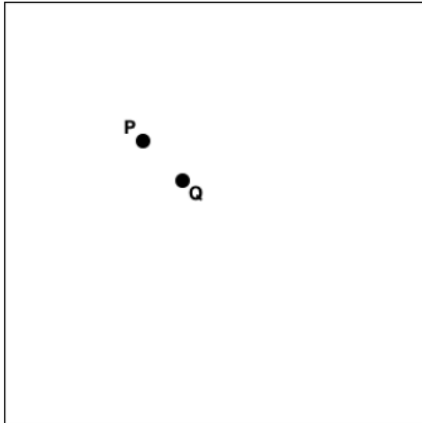
We insert a second point Q. Note that the colour changes from black to grey as the root node becomes an internal node (a leaf node can only hold a single datapoint but now holds two):

Since the root is an internal node, we have to subdivide the rectangle into four sub-rectangles and have to add four children in the corresponding quadtree. However, since both points are still located in the same sub-rectangle NW, we colour this node as grey as well in the corresponding quadtree which has now a root and four leaves:
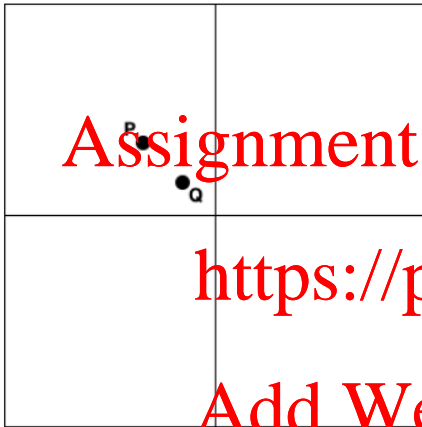
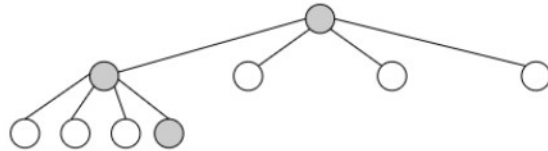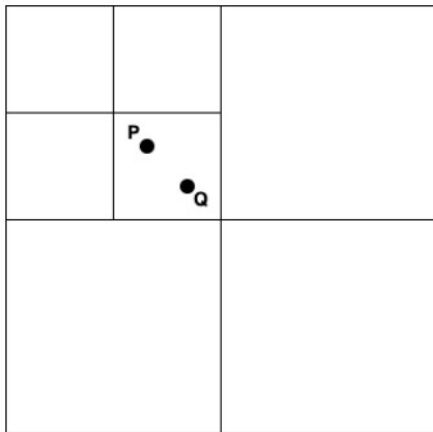Hence, we subdivide the NW rectangle further into four sub-rectangles. We add four more children nodes to the grey node that we coloured as grey in the previous step:

Since this subdivision again does not lead to P and Q being in separate rectangles, we have to add another layer to the quadtree, i.e., subdivide the rectangle that includes both points once more into four rectangles. Since P and Q are now in separate rectangles, the corresponding nodes become leaves in the quadtree coloured in black and the two other children are simply coloured white. We have now completed the update of inserting Q:
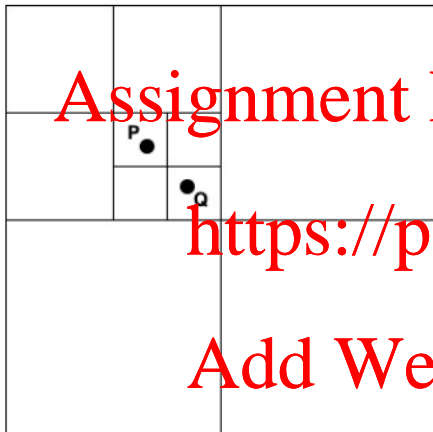
Finally, we insert a third point R. This is a simple case, because the leaf on the first level is empty, i.e., we simply colour the SE node on the first level of the quadtree as black:
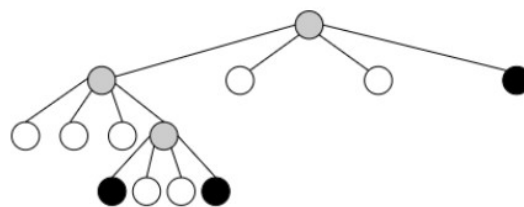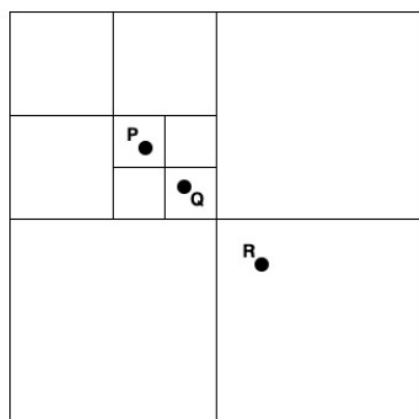
Further References

- Here is a good introduction to PR quadtrees that has some animated visualisations: link.
- You might also find this demonstration useful in deepening your understanding: link.

Your Task

Your implementation of a PR quadtree has to support a number of functions to store data points and to search for them. You will need to implement functions to support inserting data points, searching for individual datapoints and searching (returning) for all datapoints within a query rectangle.

Insertion

To insert a data point into a PR quadtree we start with the root node. If the PR quadtree is empty, i.e., the root node is coloured as white (which we represent as a pointer to NULL) and the point lies within the area that should be covered with the PR quadtree, the data point becomes the root. If the root is an internal node, we compute in which of the four children the data points lies (of course, if the data point lies outside of root rectangle, it is not inserted and an error message returned). Again, if the corresponding node is an internal node, we recursively repeat this process until we find a leaf node. If the leaf node is empty, we simply insert the point. If the leaf node is black, i.e., has already one node, we need to split the leaf node into four children, i.e., the existing black leaf node becomes a grey internal node. We repeat splitting the node until we find an internal node so that two of its children contain the existing and the newly inserted datapoint, i.e., are black leaf nodes. In other words: splitting a black leaf can also led to a recursive process.

Find (Search)

To find a datapoint (and often its associated information), we simply traverse the PR quadtree by selecting the internal child node whose rectangle contains the location of the datapoint. The search stops once we have reached a leaf node. If the leaf node has stored the datapoint, we return its associated information; otherwise the search simply reports that the datapoint is not stored in the PR quadtree.

Range (Window) Query

An important query is to report all datapoints whose locations are contained in a given query rectangle. This query is typically called a range or window query. To run range query on a quadtree, we have the following recursive process: we start the root and check if the query rectangle overlaps with the root. If it does, we check which children nodes also overlap with query rectangle (this could be all of them for a centred query rectangle). Then we repeat this procedure for all internal children overlapping with the query rectangle. The recursion stop once we have reached the

leaf level. We return the datapoints from all black leaf nodes whose datapoints are located within the query rectangle.

Implementation Details

To implement your own PR quadtree, you will need to set up some data structures and functions, you will likely need a number of structures and helper functions, these are not required and you do not need to follow the naming style or function as described, but you will likely find them useful in your implementation. As a minimum you will likely have the following data structures:

- `point2D` that stores the location of datapoint;
- `rectangle2D` that specifies a rectangle given an bottom-left 2D point and a upper-right 2D point;
- `dataPoint`, which is a structure that stores the location and associated information (i.e., the footpath information);
- `quadtreeNode`, which stores a 2D rectangle and 4 pointers referring to the four children SW, NW, NE and SE.

You will also likely need the following functions:

- `inRectangle`: tests whether a given 2D point lies within the rectangle and returns 1 (TRUE) if it does. The function takes the point and the rectangle; note that the convention is that points are within a rectangle if they are on lower and right boundary but not on the top or left boundary. This enables us to construct a well-defined partition of the space if points lie on boundaries of the quadtree node rectangle.
- `rectangleOverlap`: tests whether two given rectangles overlap and returns 1 (TRUE) if they do.
- `determineQuadrant`: given a rectangle and point, returns the quadrant of the rectangle that the point lies in. The convention is `0=SW`, `1=NW`, `2=NE`, `3=SE` for the quadrant. You may like to implement helper functions which also select relevant pointers from the `quadtreeNode` given a quadrant. You may find writing an `enum` quadrant useful for implementing and using this function.
- `addPoint`: this function will add a datapoint given with its 2D coordinates to the quadtree. You will need to setup the logic discussed before.
- `searchPoint`: tests whether a datapoint given by its 2D coordinates lies within the rectangle and returns the datapoint (and its stored information) if it does and NULL otherwise.
- `rangeQuery`: takes a rectangle as argument and returns all datapoints in the PR quadtree whose coordinates lie within the query rectangle.

ℹ Tip: if you have begun implementing and are stuck, check this page, you will likely need one of these functions.

Stage 3 - Supporting Point Region Queries

In Stage 3, you will implement the basic functionality for a quadtree allowing the lookup of data by longitude (x) and latitude (y) pairs.

Your `Makefile` should produce an executable program called `dict3`. This program should take seven command line arguments.

1. The first argument will be the *stage*, for this part, the value will always be 3.
2. The second argument will be the filename of the *data file*.
3. The third argument will be the filename of the *output file*.
4. The fourth and fifth argument specify the `x`, `y` co-ordinate pair of the bottom-left corner of the root node area, with the first value referring to the *longitude* (`x`) and the second value referring to the *latitude* (`y`).
5. The sixth and seventh argument specify the top-right corner of the root node area, following the same convention.

Your program should:

- Just as in Assignment 1, read data from the data file specified in the second command line argument. This may be stored unchanged from `dict1` or `dict2` in earlier implementations.
- Construct a quadtree from the stored data.
- Interpret and store the fourth, fifth, sixth and seventh arguments as `long double` values. The `strtold` function should be used to achieve this.
- Accept co-ordinate pairs from *stdin*, search the constructed quadtree for the point region containing the co-ordinate pair and print all matching records to the *output file*. You may assume that all queries will be terminated by a new line. You should interpret these values as `double` values.
- In addition to outputting the record(s) to the output file, the list of quadrant directions followed from the root until a matched point region is found should be output to *stdout*.
- Your approach should insert each footpath's (`start_lon`, `start_lat`) and (`end_lon`, `end_lat`) pairs into the quadtree, allowing the footpath to be found from its start *or* end point.
- Where multiple footpaths are present in the found point region, footpaths should be printed in order of `footpath_id`.

The quadrants in relation to the dataset are:

SW	($<$ longitude, $<$ latitude)
NW	($<$ longitude, $>$ latitude)
NE	($>$ longitude, $>$ latitude)
SE	($>$ longitude, $<$ latitude)

> ⚠ Your approach for the PR Quadtree must use `long double` values for representing regions. This 16-byte variant of `float` is more precise than the 8-byte `double` variant. No changes to your record implementation are needed, that is, you do not need to change your record struct's types to `long double`. (You should be using `double` though!)

For testing, it may be convenient to create a file of queries to be searched, one per line, and redirect the input from this file. Use the UNIX operator `<` to redirect input from a file.

Example Execution

An example execution of the program might be:

```bash
1  make -B dict3
2  # ./dict3 stage datafile outputfile start_longitude start_latitude end_longitude en
3  ./dict3 3 dataset_2.csv output.txt 144.969 -37.7975 144.971 -37.7955
```

followed by typing in the queries, line by line, and then ending input once all keys are entered or:

```bash
1  make -B dict3
2  # ./dict3 stage datafile outputfile start_longitude start_latitude end_longitude en
3  ./dict3 3 dataset_2.csv output.txt 144.969 -37.7975 144.971 -37.7955 < queryfile
```

Example Output

This is an example of what might be output to the *output file* after three queries:

```
144.97056424489568 -37.796155887263744
--> footpath_id: 27665 || address: Palmerston Street between Rathdowne Street and Drummond Street || clue_sa:
144.96941668057087 -37.79606116572821
--> footpath_id: 27665 || address: Palmerston Street between Rathdowne Street and Drummond Street || clue_sa:
144.95538810397605 -37.80355555400948
--> footpath_id: 19458 || address: Queensberry Street between Capel Street and Howard Street || clue_sa: Nort
```

With the following output to *stdout*:

```
144.97056424489568 -37.796155887263744 --> NE SW NE NE
144.96941668057087 -37.79606116572821 --> NE SW NE NW
144.95538810397605 -37.80355555400948 --> SW NW SE
```

这个测试指令有误

Stage 4 - Supporting Range Queries

In Stage 4, you will add the additional functionality to your quadtree to support range queries given by `(x, y)` co-ordinate pairs.

Your `Makefile` should now also produce an executable program called `dict4`. This program should take seven command arguments, similar to Stage 3, with only the expected value of the first argument changing:

1. The first argument will be the *stage*, for this part, the value will always be 4.
2. The second argument will be the filename of the *data file*.
3. The third argument will be the filename of the *output file*.
4. The fourth and fifth argument specify the `x`, `y` co-ordinate pair of the bottom-left corner of the root node area, with the first value referring to the *longitude* (`x`) and the second value referring to the *latitude* (`y`).
5. The sixth and seventh argument specify the top-right corner of the root node area, following the same convention.

Your `dict4` program should:

- Just as Stage 3, read in the dataset, store it in a dictionary and construct a quadtree on the stored data.
- For Stage 4, you should accept sets of pairs of co-ordinate `long double` type values from *stdin*, and efficiently use the quadtree to find all footpaths which are within the bounds of the query. You may assume that no blank lines will be present in the queries.
- Output to `stdout` should include all directions searched in order, with each branch potentially containing points within the query bounds fully explored before proceeding to the next possible branch. Where multiple directions are possible, quadrants must be searched in the order `SW`, `NW`, `NE`, `SE`. Each direction must be separated by a space. The definitions of each quadrant are given in the table below.
- Similar to Stage 3, where multiple footpaths are returned by the range query, these should be sorted by `footpath_id`. Output each footpath only once for each query, even if both its "start" and "end" points both occur in the searched region.

Example Execution

An example execution of the program might be:

```bash
1  make -B dict4
2  # ./dict4 stage datafile outputfile start_longitude start_latitude end_longitude en
3  ./dict4 4 dataset_2.csv output.txt 144.968 -37.797 144.977 -37.79
```

followed by typing in the queries, line by line, and then ending input once all keys are entered or:

```bash
1  make -B dict4
2  # ./dict4 stage datafile outputfile start_longitude start_latitude end_longitude en
3  ./dict4 4 dataset_2.csv output.txt 144.968 -37.797 144.977 -37.79 < queryfile
```

Example Output

This is an example of what might be output to the *output file* after three queries:

```
144.968 -37.797 144.977 -37.79
--> footpath_id: 27665 || address: Palmerston Street between Rathdowne Street and Drummond Street || clue_sa:
--> footpath_id: 29996 || address:  || clue_sa: Carlton || asset_type: Road Footway || deltaz: 0.46 || distan
144.9678 -37.79741 144.97202 -37.79382
--> footpath_id: 27665 || address: Palmerston Street between Rathdowne Street and Drummond Street || clue_sa:
144.973 -37.795 144.976 -37.792
--> footpath_id: 29996 || address:  || clue_sa: Carlton || asset_type: Road Footway || deltaz: 0.46 || distan
```

With the following output to *stdout*:

```
144.968 -37.797 144.977 -37.79 --> SW SW SE NE SE
144.9678 -37.79741 144.97202 -37.79382 --> SW SW SE
144.973 -37.795 144.976 -37.792 --> NE SE
```

The dataset used in this project is a subset of the Footpath Steepness dataset combined with data from the Small Areas for Census of Land Use and Employment (CLUE) dataset. This dataset has been processed to simplify the geometric polygon data into additional attributes `clue_sa`, `start lon`, `start lat`, `end lon`, `end lat`, which can be used to approximate the footpath as a line on the map.

> ℹ The processed dataset can be found in the **Dataset Download** slide. You aren't expected to perform this processing yourself.

The provided dataset has the following 19 fields:

```
footpath_id   - The row number for this footpath in the original full dataset. (integer)
address       - A name describing the location of the footpath. (string)
clue_sa       - The CLUE small area the footpath is in. (string)
asset_type    - The name of the type of footpath. (string)
deltaz        - The change in vertical distance along the footpath. (double)
distance      - The length of the footpath (full geometry) in metres. (double)
gradelin      - The percentage gradient of the footpath (full geometry). (double)
mcc_id        - The id number identifying the footpath. (integer)
mccid_int     - A second number identifying the road or intersection the footpath borders. (integer)
rlmax         - The highest elevation on the footpath. (double)
rlmin         - The lowest elevation on the footpath. (double)
segside       - The side of the road which the footpath is on. (string)
statusid      - The status of the footpath. (integer)
streetid      - The ID of the first street in the Address. (integer)
street_group  - The footpath_id of one of the footpaths connected to this footpath without a gap. (integer)
start_lat     - The latitude (y) of the starting point representing the line approximation of the footpath. (d
start_lon     - The longitude (x) of the starting point representing the line approximation of the footpath. (
end_lat       - The latitude (y) of the ending point representing the line approximation of the footpath. (dou
end_lon       - The longitude (x) of the starting point representing the line approximation of the footpath. (
```

The fields `address`, `clue_sa`, `asset_type` and `segside` are strings, and may be zero-length and could contain spaces. You may assume none of these fields are more than 128 characters long. The fields `footpath_id`, `mcc_id`, `mccid_int`, `statusid`, `streetid` and `street_group` are integers, you may assume they are always specified and present even though a value of 0 signifies the information is not present or not applicable.

This data is in CSV format, with each field separated by a comma. For the purposes of the assignment, you may assume that:

- the input data are well-formatted,
- input files are not empty,
- input files include a header,
- fields always occur in the order given above, and that
- the maximum length of an input record (a single full line of the CSV) is 512 characters.

Where a string contains a comma, the usual CSV convention will be followed that such strings will be delimited with a quotation mark (always found at the beginning and end of the field) — these quotes should not be preserved once the data is read into its field. You may assume no quotes are present in the processed strings when they are stored in your dictionary.

The fields `address`, `clue_sa`, `asset_type` and `segside` are strings, and may be zero-length and could contain spaces. You may assume none of these fields are more than 128 characters long. The fields `footpath_id`, `mcc_id`, `mccid_int`, `statusid`, `streetid` and `street_group` are integers, you may assume they are always specified and present, even though a value of 0 signifies the information is not present or not applicable.

This data is in CSV format, with each field separated by a comma. For the purposes of the assignment, you may assume that:

- the input data are well-formatted,
- input files are not empty,
- input files include a header,
- fields always occur in the order given above, and that
- the maximum length of an input record (a single full line of the CSV) is 512 characters.

Where a string contains a comma, the usual CSV convention will be followed that such strings will be delimited with a quotation mark (always found at the beginning and end of the field) — these quotes should not be preserved once the data is read into its field. You may assume no quotes are present in the processed strings when they are stored in your dictionary.

Smaller samples of these datasets can be found in the **Dataset Download** slide.

Requirements

The following implementation requirements must be adhered to:

- You must write your implementation in the C programming language.
- You must write your code in a modular way, so that your implementation could be used in another program without extensive rewriting or copying. This means that the dictionary operations are kept together in a separate `.c` file with its own header file, else separate from the main program, and other distinct modules are similarly separated into their own sections, requiring as little knowledge of the internal details of each other as possible.
- Your code should be easily extensible to multiple dictionaries. This means that the functions for interacting with your dictionary should take as arguments not only the values required to perform the operation required, but also a pointer to a particular dictionary, to search dictionary variable1.
- Your implementation must read the input file once only.
- Your program should store strings in a space-efficient manner. If you are using `malloc()` to create the space for a string, remember to allow space for the final end of string character, `'\0'` (`NULL`).
- Your program should store its data in a space-efficient manner. If you construct an index, this index should not contain information it does not need.
- Your approach should be reasonably *time efficient*.
- You may assume the quadtree only needs to be built after all records are read for simplicity.
- You may assume you will not be provided with points outside the specified region.
- A full Makefile is not provided for you. The Makefile should direct the compilation of your program. To use the Makefile, make sure it is in the same directory as your code, and type `make dict3` and `make dict4` to make the dictionary. You must submit your Makefile with your assignment.

> ℹ️ **Hints:**
> • If you haven't used `make` before, try it on simple programs first. If it doesn't work, read the error messages carefully. A common problem in compiling multifile executables is in the included header files. Note also that the whitespace before the command is a tab, and not multiple spaces.
> • It is not a good idea to code your program as a single file and then try to break it down into multiple files. Start by using multiple files, with minimal content, and make sure they are communicating with each other before starting more serious coding.

Assessment

There are a total of 15 marks given for this assignment.

Your C program will be marked on the basis of accuracy, readability, and good C programming structure, safety and style, including documentation (1 mark). Safety refers to checking whether opening a file returns something, whether `malloc()`s do their job, etc. The documentation should explain all major design decisions, and should be formatted so that it does not interfere with reading the code. As much as possible, try to make your code self-documenting by choosing descriptive variable names.

⚠️ It is common to lose marks for modularity and efficiency because the **Requirements** slide was not adequately read, make sure to double check requirements at intervals during the writing of your program and to check Ed regularly.

The remainder of the marks will be based on the correct functioning of your submission.

⚠️ Note that unlike the first assignment, passing test cases will not guarantee you marks - for example, your approach *should be reasonably time efficient* (such as sorting and uniqueness checks). Passing test cases is approximately aligned with demonstrated functionality but does not directly evaluate efficiency of code or careful adherence to the specification's Implementation Details, Requirements or Task requirements.

```
| Marks | Task                                                                              |
|-------|-----------------------------------------------------------------------------------|
|   7   | Stage 3 implementation is correct and reasonably efficient.                       |
|   2   | Stage 3 memory correctly freed and is free of errors.                             |
|   3   | Stage 4 implementation is correct and reasonably efficient.                       |
|   2   | Stage 4 memory correctly freed and is free of errors.                             |
|   1   | Program style consistent with Programming Style slide, code sufficiently modular. Memory |
|       | allocations and file opens checked.                                               |
```

Assignment Project Exam Help

https://powcoder.com

Add WeChat powcoder