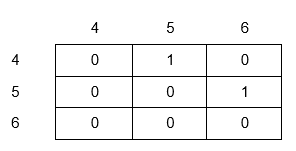
## PKB Tables’ Data Structure

In this section we will show our unique data structure and how do we come up with that solution.

At first, we did try two common ways to save PKB Data Tables, which is by table and by 2D vector. However, each of this method has its own drawback.

**Table**

|  |
| --- |
| Pros:  -Relationship checking, like “is 4 parent of 6”, can be done in O(1). |
| Cons:  -Retrieving all data relating to a key, like “get all parent of 4”, will be O(N).  -When parsing source code, we cannot fix an initial size for this table from the beginning and have to resize every time the contained data reached a limit. This resize process is O(N2). |

**2D Vectors**

|  |
| --- |
| Pros:  -Retrieving all data relating to a key, like “get all parent of 4”, will be O(1). |
| Cons:  -Relationship checking, like “is 4 parent of 6”, can be done in O(N). |

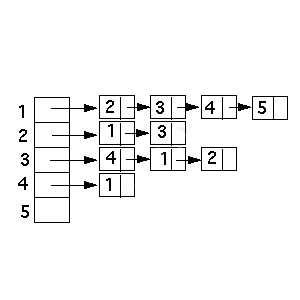


Figure 1: *Advantages and Drawbacks of each common method*

So the question is How to take the advantages of both data structure. One solution can be using both of these data structure and call them accordingly to the query. For example, use table for query “is 4 parent of 6” and use 2D vector for “get all parent of 4”. But then there will be too many data be saved for each relationship.

For example, Parent relationship will have:

-2 tables to check isParent() and isParentStar(),

- 2 2D vectors to store, child

As a result, totally each relationship have 6 tables and 2D vectors which wastes space, increase code written and hard to maintain. Therefore we come up with something in the middle that balance those advantages and drawbacks from both trivial data structures above. We build our own generic data structure that can fit to all relationships in this project, which are called “ListTable” and “MapTable”.

Figure 2: ADT used for PKB

In detail, there is a base class call BaseTable to which ListTable and MapTable inherent.

ListTable is simple, which contain only a vector in it. ListTable is used to store non-relationship data like Procedures, Variables, Constants, Statements.

MapTable is more complicated which used c++ map data structure. MapTable is also generic but only accept key and value of the same data type because of the requirement of “star” relationship. MapTable is used for relationship data like follow, modify, use, parent, call, next, contain, sibling.

**Benefits:**

Time Complexity Improvement: the time complexity below still can be reduce is we choose to save more information or use all suitable data structure for each kind of query, but we choose to minimum the code written and the data saved with a reasonable time complexity.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Is x parent of y | Is x parent\* of y | Get all child of x | Get child\* of x |
| Single original table | O(1) | O(1) | O(N) | O(N) |
| 2D Vector | O(N) | O(N) | O(1) | O(1) |
| **new MapTable** | O(log(N)) | O(log(N)) | O(log(N)) | O(log(N)) |

Code Maintenance and Extension: By implement all data table at a single place, developer can easily change and extend without doing a lot of duplicating works in more than ten table. This reduce the change of making mistake. Testing can also be easier when tester don’t have to write unit test case for every single tables.

To give an example, the implementation of the Next Table could look like as follows.

MapTable <int> NextTable;

keyValue (Map of type <int, vector<int>>)

Possible value pairs: (1,2), (2,3), (3, 4), (4, 5)

valueKey (Map of type <int, vector<int>>)

Possible value pairs: (2,1), (3,2), (4, 3), (5, 4)

keyValueStar (Map of type <int, vector<int>>)

Possible value pairs: (1, [2, 3, 4, 5])

### Next/Next Star

Next is saved normally as all other relationship.

The tricky part here is that we cannot save Next Star so it will be O(N) if we do a trivial CFG travel. However we managed to reduce this to O(1) by apply some strategies.

In different branch of an IF?

In the same WHILE loop?

Is stmt 1 < stmt2

This is the logic:

* If 2 statements belong to the same WHILE loop then they will lead to each other anyway.
* If not, see if they belong to the same if, and if they on the same or different branch of that if. If they are on two different branch then we can be sure that they will never meet.
* Finally we check if statement 2 if appear after statement 1 or not, if it is, then they are next star.

The information of common WHILE and IF parent is saved beforehand so the complexity of this algorithm is O(1).

### Affects/Affects Star

Affect and Affect\* is purely base on CFG travel. On the path, if any assignment use the original modified variable, that statement will be considered “Affected” by the origin of the path. The Algorithm will stop travel when the original modified variable being modified by a statement on the path. By doing this, we warranty the conditions that lead to an Affect relation:

* 2 statements can lead to each other in the CFG
* Variable that be modified in statement 1 is used in statement 2
* On the path, that variable isn’t modified by any statement.

An example is the graph beside, to get statements that being affected by 1, we will travel along the CFG, 2 use x then we add 2 in the list of result, 3 modifies x then we stop travel at 3 and not go to 4.

Affect\* is similar to affect except that we will maintain a list of modified variables, when we meet a statement that modifies a variable in this list, instead of stop traveling, we remove that variable from the list. In addition, when meet a statement that uses one of the modified variable, we add the modified variable of that statement to the list of modified variable. Beside that we also apply some minor strategies to dealing with WHILE loop because it can contain some tricky cases sometimes.