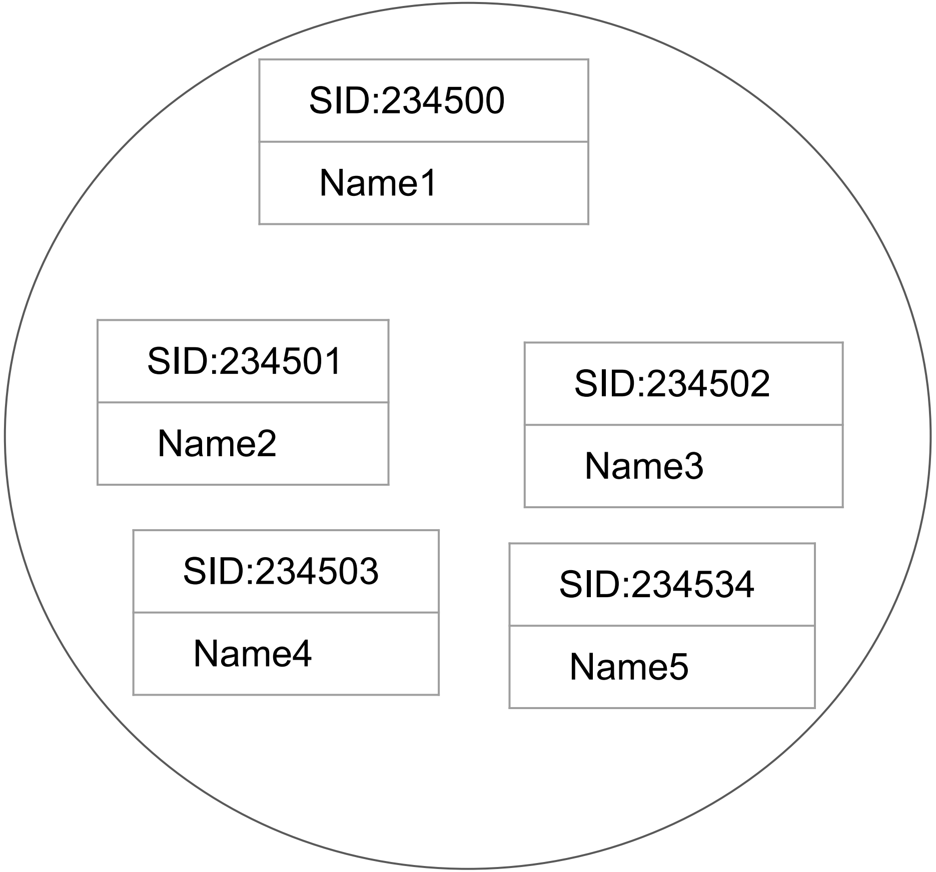
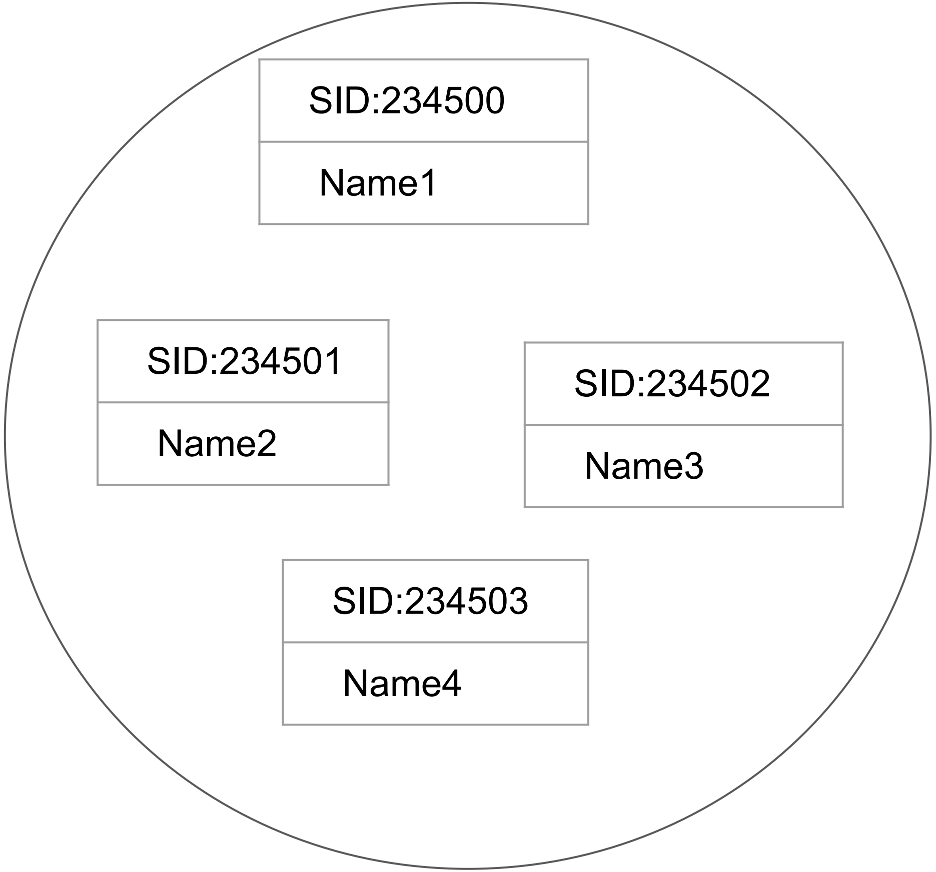
Student Records - Student IDs



Use Case: Store the records in a such way that we can fetch records by IDs(which is the **key**).

How do you do it?

Student Records - Student IDs



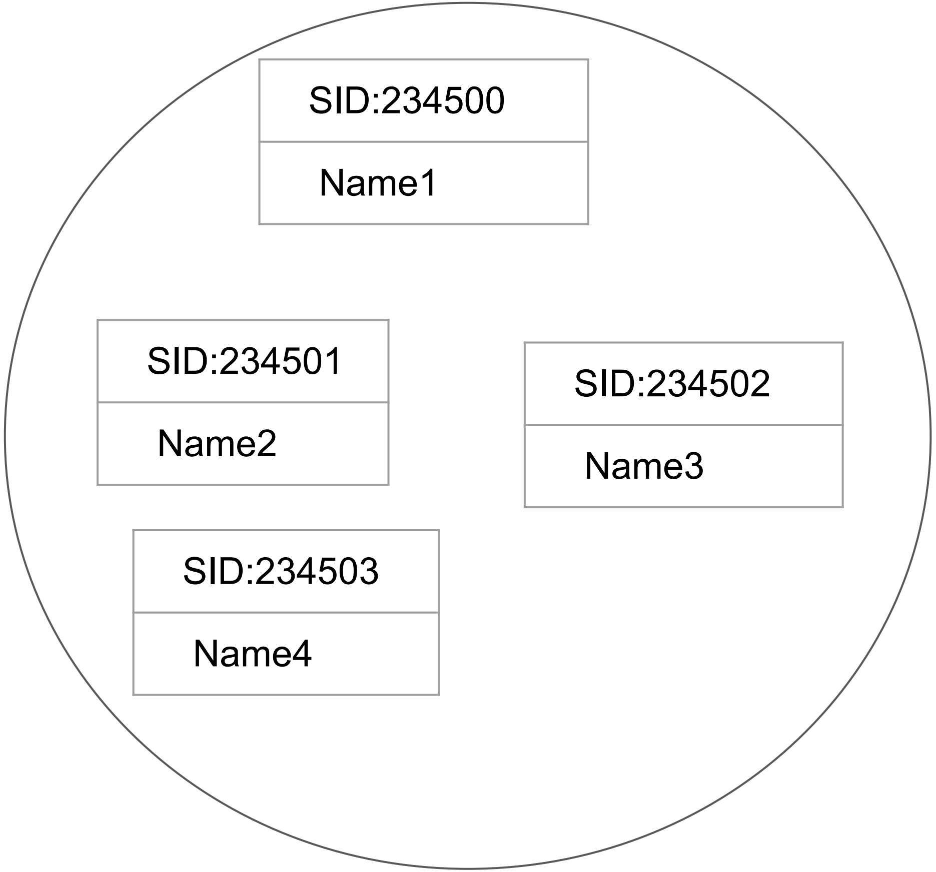
Array?
Using array, we access the elements by index.

Index	struct
0	key:234500 Data:Name1
1	key:234501 Data:Name2
2	key:234502 Data:Name2
3	key:234503 Data:Name3

Student Records - Student IDs

Using **Array**

To find a student’s records, we have to iterate through the array and find the record for which the key matches.

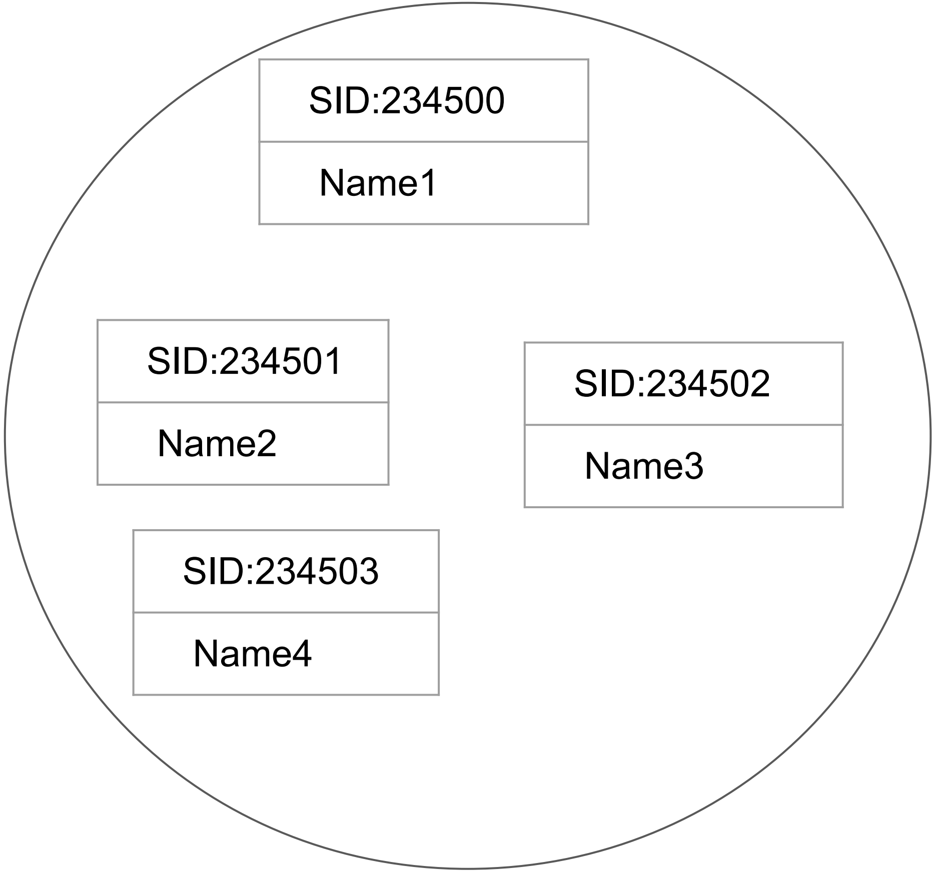


Index	struct
0	key:234500 Data:Name1
1	key:234501 Data:Name2
2	key:234502 Data:Name2
3	key:234503 Data:Name3

Student Records - Student IDs

Using **Array**

To find a student's records, we have to iterate through the array and find the record for which the key matches.

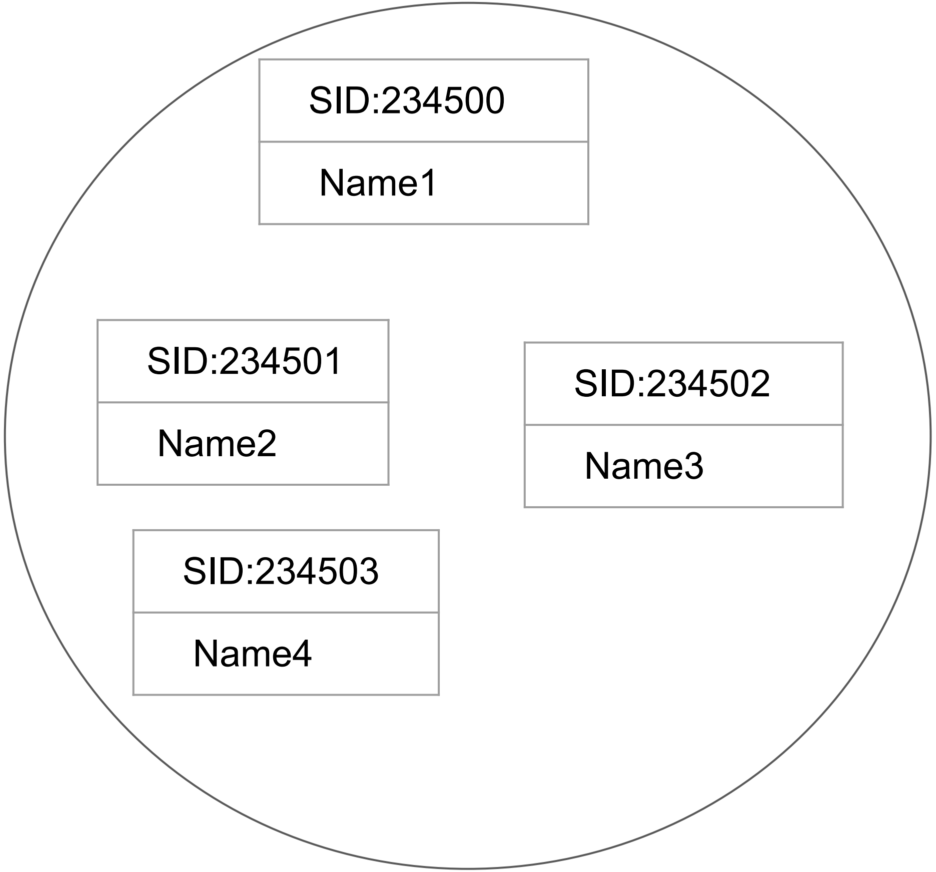


Index	struct
0	key:234500 Data:Name1
1	key:234501 Data:Name2
2	key:234502 Data:Name2
3	key:234503 Data:Name3

Student Records - Student IDs

Using **Array**

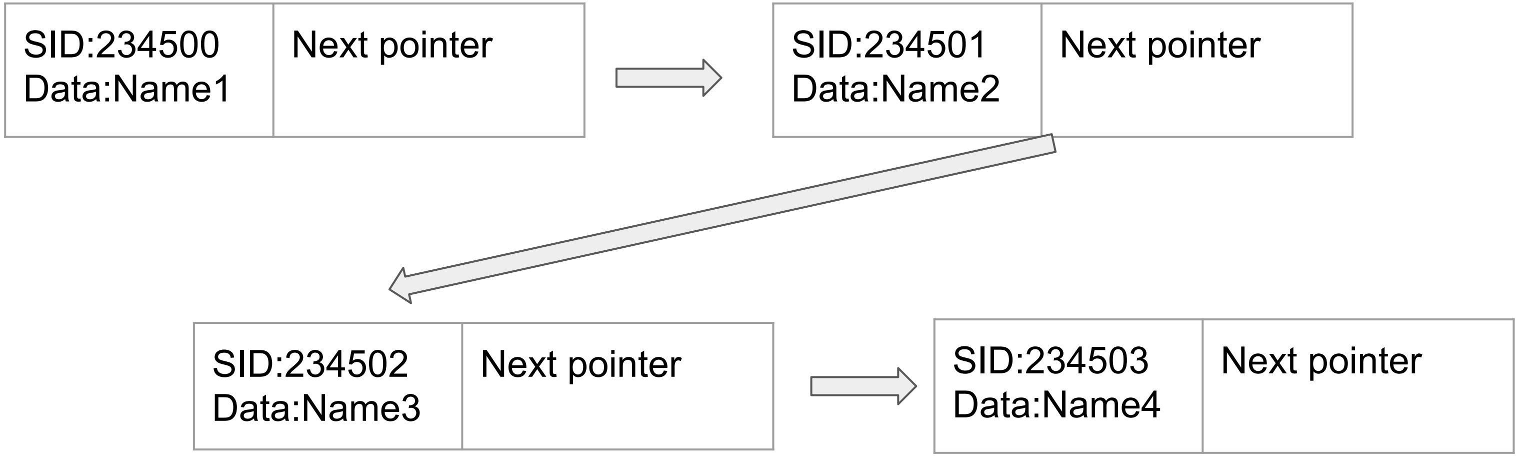
Time taken is proportional to **number of students(size of array)**



Index	struct
0	key:234500 Data:Name1
1	key:234501 Data:Name2
2	key:234502 Data:Name2
3	key:234503 Data:Name3

Student Records - Student IDs

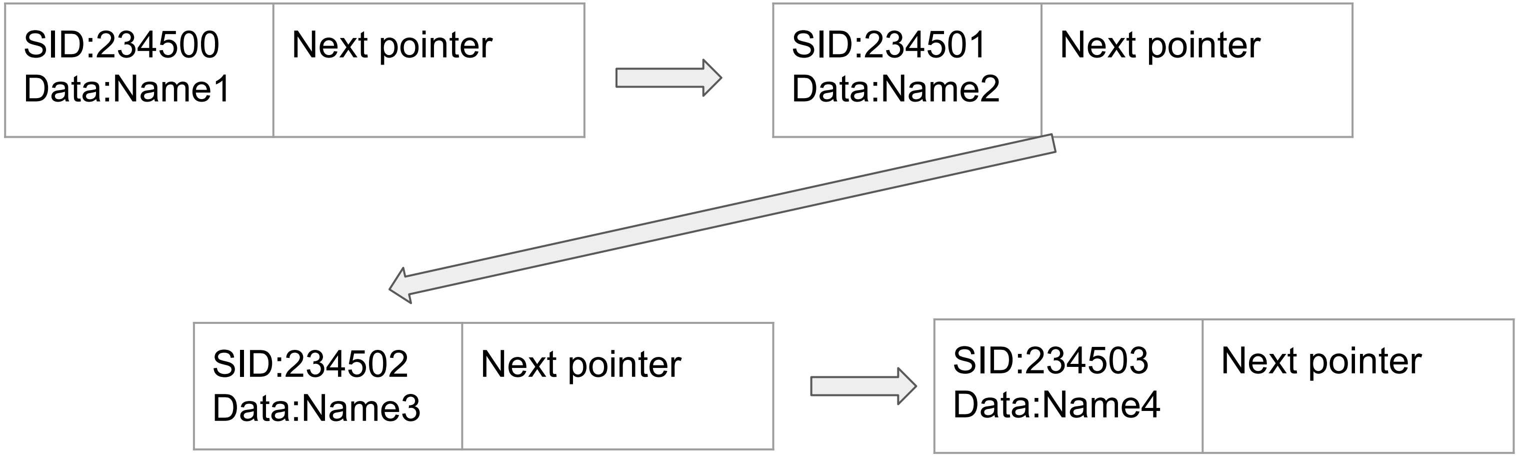
Using **Linked List**



Student Records - Student IDs

Using Linked List

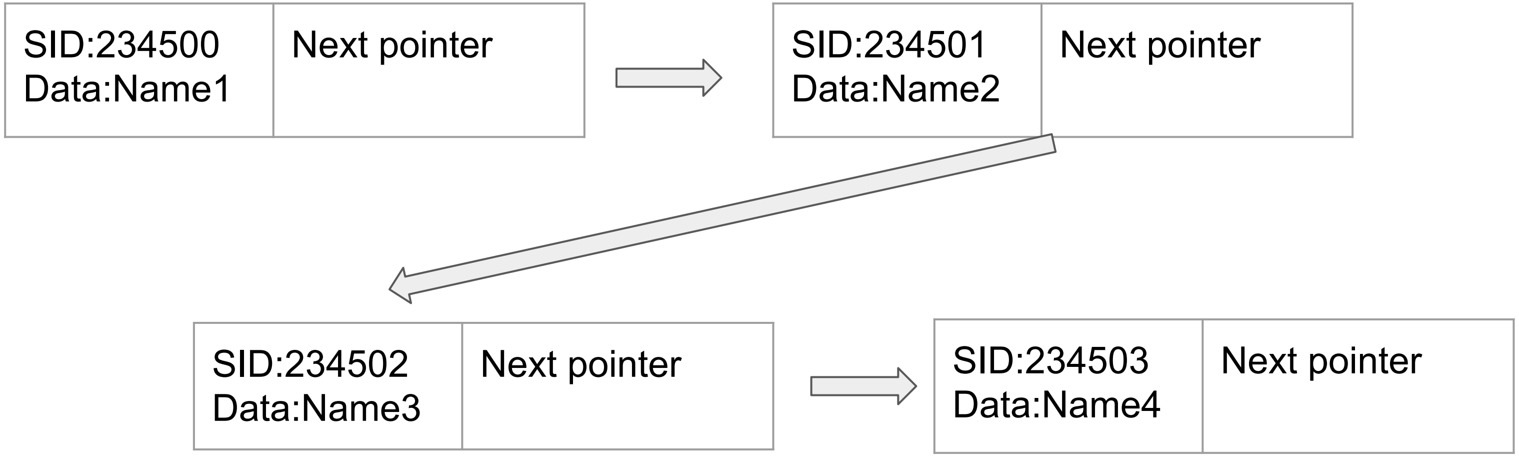
To find a student's records, we have to iterate through the linkedlist and find the record for which the key matches.



Student Records - Student IDs

Using **Linked List**

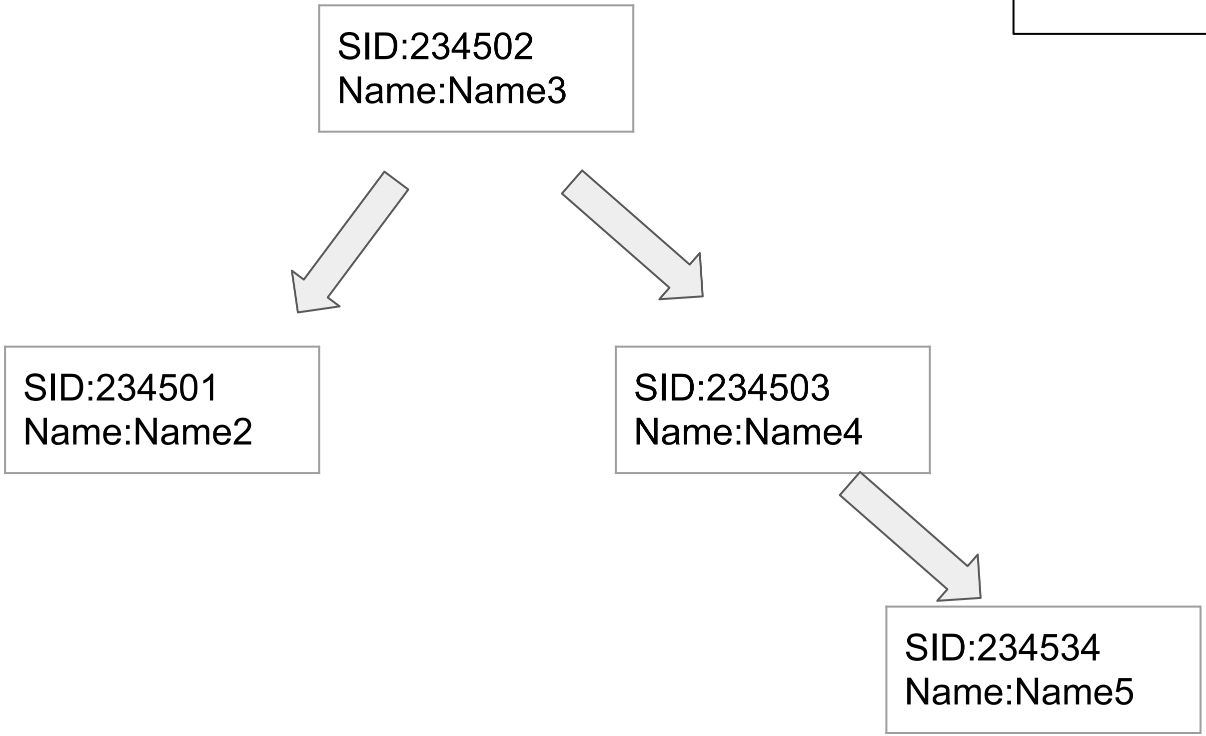
Time taken is still proportional to size of the number of students.



Student Records - Student IDs

Using **BST**

Time taken is proportional to the height of the tree which will be logarithmic.



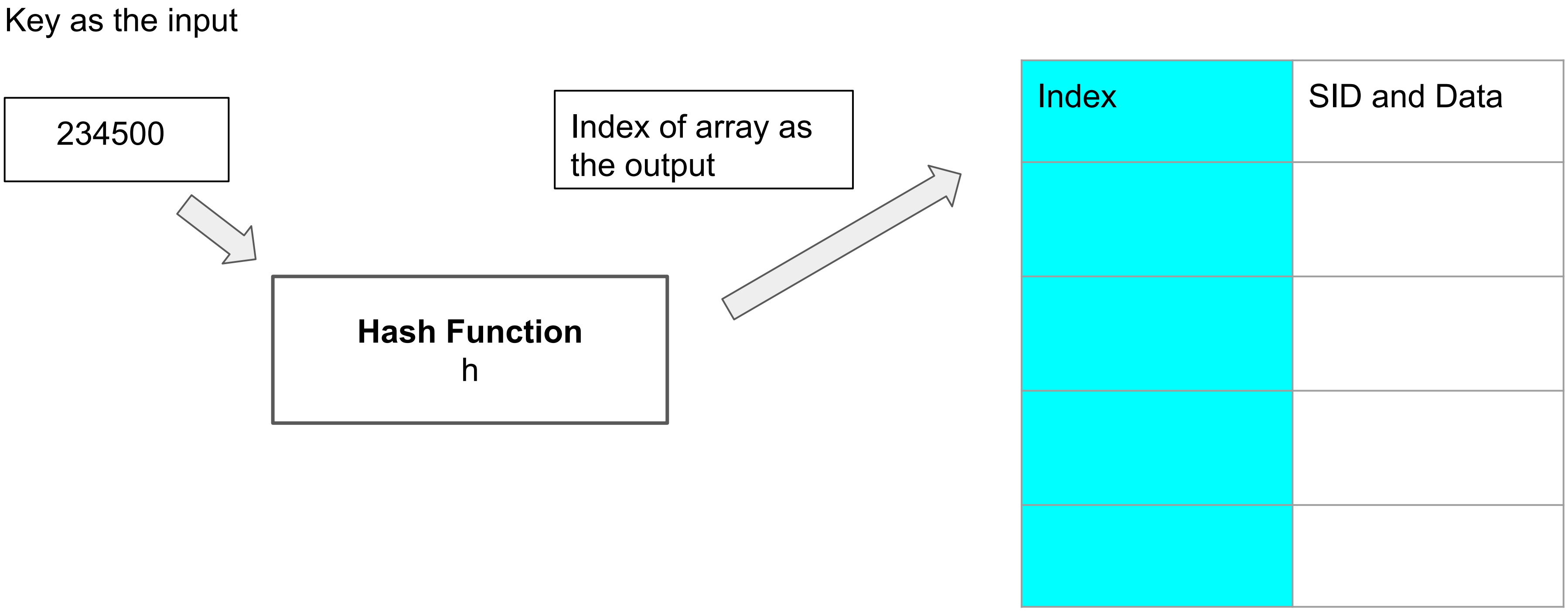
What if I use the key as the index?

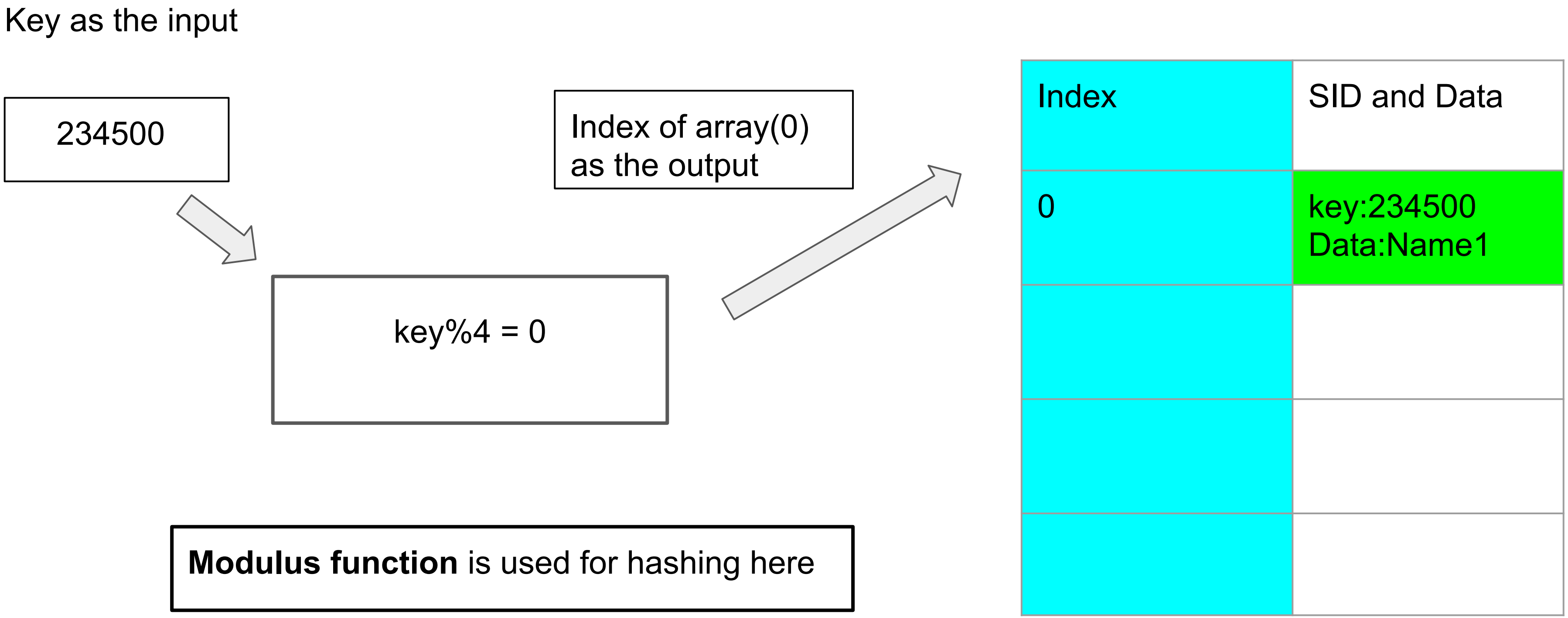
Lots of unused space.

We have just 4 records, and we end up having an array of size which is proportional to 10^6. (Since there are 6 digits)

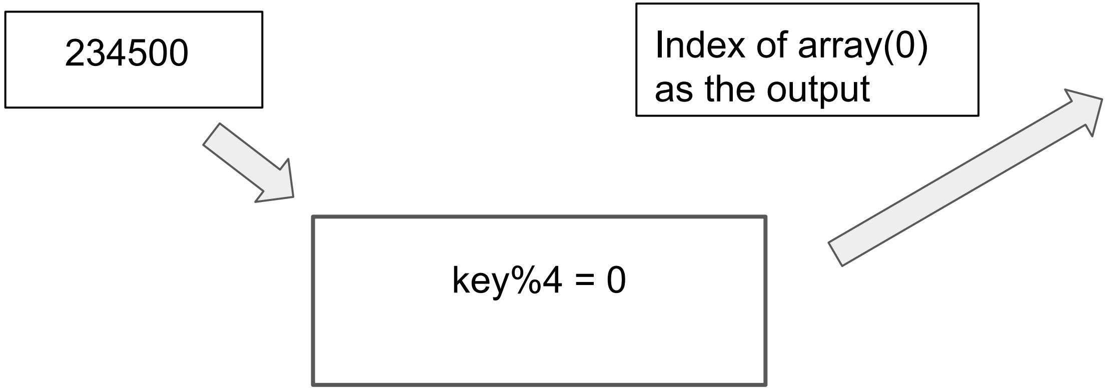
Index	Data
0	EMPTY
1	EMPTY
2	EMPTY

234500	Data:Name1
234501	Data:Name2
234502	Data:Name2
234503	Data:Name3


Key as the input
234500
Hash Function
h
Index of array as the output
Index
SID and Data


Key as the input
234500
key%4 = 0
Index of array(0)
as the output
Index
SID and Data
0
key:234500
Data:Name1
Modulus function is used for hashing here

Key as the input



Index	SID and Data
0	key:234500 Data:Name1

This takes constant time to access or insert data using a key on average.

New key

234504

key%4 = 0

Index of array(0) as the output

Index	SID and Data
0	key:234500 Data:Name1

Here we are inserting an element which returns the index **0**. **But there was already an element for this index**. This is called as a **collision**.

Chaining- We make each element of the array a linked list.
Every element in the array points to the **head of a linked list**.

New key

234504

key%4 = 0

Index of array(0)
as the output

Index	SID and Data
0	key:234504 Data:Name5 **nextPointer**

key:234500
Data:Name1
nextPointer

Resolving collisions.

Table[index]
This fetches the head
of the linked list

Trees:

Solving problems in Trees using recursion

Types of traversals:

Pre, Inorder, Postorder

When solving any problem using recursion, get the base case/edge cases right.

Visualise everything and see if your code will work on the test cases provided by doing a dry run on a piece of paper.
Go through all the recitations/assignments/quizzes(if accessible). Problems will be similar.

Binary Search Tree:

- Insert,Search, Delete(get all the cases right) operations
- Make use of the BST property well when performing any operation.
- When updating the BST(insertion/deletion) , the BST property has to remain intact.
- When you can’t use the given function directly for recursion , make your helper function.
- Understand the time complexities for each of the operations.

Sample problems from piazza and practice midterm :

1. In a Binary search tree, compute sum of all the nodes which has exactly one child (either left or right child but not both).
2. Given a Binary Search Tree, return the sum of values of all the nodes that fall within a range [min, max] inclusive.

Graphs:

- Representation - using adjacency matrix and lists.
- Unweighted and weighted graphs.
- Directed and undirected graphs.
- Traversal/Search algorithms - Breadth First /Depth First.
- Implementation of BFT / DFT - Make sure you take care of marking visited flags, enqueuing and dequeuing nodes, updating distance.
- BFT- shortest path for unweighted graph from a source to destination.
- Finding connected components using BFT or DFT. (Understand and see how these traversal algorithms are used to solve the problem)
- Dijkstra’s - for finding shortest path in a weighted graph
- Time complexity for the above algorithms

Hash tables:

- Conceptual questions
- Have a thorough understand on how elements are inserted into the hash table. Practice the quiz problems well.
- Try out different hash functions, different collision resolution strategies(chaining/linear probing/double hashing etc) and see how they work.
- Understand the time complexities for various operations/ load factor and other concepts.

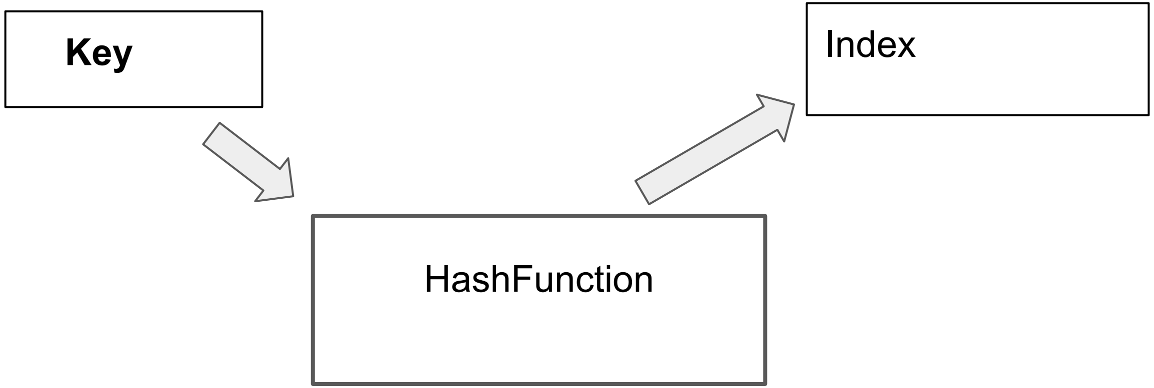
Search in a hash table

```
node* HashTable::searchItem(int key)
{

        //1. Call the hash function to compute the index.

        //2. Retrieve the node in a temp variable.

        //3. Iterate through the linked list till you find the node and return it.
}
```



Search in a hash table

```
node* HashTable::searchItem(int key)
{

    //1. Call the hash function to compute the index.

    //2. Retrieve the node in a temp variable. (This is the head of the linked list)

    //3. Iterate through the linked list till you find the node and return it.
}
```

	Index	SID and Data
Head	0	key:234508 Data:Name9 **nextPointer**

key:234504
Data:Name4
nextPointer

key:234500
Data:Name1
nextPointer

Search in a hash table

```
node* HashTable::searchItem(int key)
{

    //1. Call the hash function to compute the index.

    //2. Retrieve the node in a temp variable. (This is the head of the linked list)

    //3. Iterate through the linked list, till you find the node and return it.
}
```

This is the node of interest

	Index	SID and Data
Head	0	key:234508 Data:Name9 **nextPointer**

key:234504
Data:Name4
nextPointer

key:234500
Data:Name1
nextPointer

Insert in a hash table

```
node* HashTable::insertItem(int key)
{
	// 1. Search for the key.(Call search function).



	// 2. If key is found, then print duplicate.



	// 3. If key not found, create a node and then insert at head of linked list.



}
```