# HYBRID DATA STRUCTURE

## RADIX HEAP

AMRITA VISHWA VIDYAPEETHAM

| | | |
|---|---|---|
| 1 | ARAVINDH | CB.EN.U4CSE21406 |
| 2 | RAMNARESH | CB.EN.U4CSE21447 |
| 3 | SURYA | CB.EN.U4CSE21461 |
| 4 | GANESH | CB.EN.U4CSE21426 |
| 5 | SHREYAS | CB.EN.U4CSE21455 |

# Introduction

## Benefits of Hybrid Data Structures:

As the name implies, hybrid data structures are data structures that mix the traits and features of several different data structures. They are created to take advantage of each component data structure's advantages, enhancing performance, effectiveness, or functionality for certain use cases. Hybrid data structures can be modified to meet needs and strike a balance among different trade-offs. Here are some benefits of using hybrid data structures:

**Efficiency gains**: When compared to traditional data structures, hybrid data structures perform specific operations, such insertion, deletion, and searching, more efficiently.

**Customized for certain use cases**: Hybrid data structures can be altered to satisfy particular use cases' needs. It is feasible to optimize the operations that are often carried out in a specific context by choosing and combining suitable data structures. This may lead to considerable performance gains for jobs or algorithms.

**Memory usage**: By using several data structures with varied memory footprints, hybrid data structures can offer greater memory utilization. When there are few elements, for instance, a hybrid structure might store them in a compact array; however, as the number of components increases, it might transition to a tree-based structure, which is more effective. This enables flexible scenario support and economical memory utilization.

**Trade-off between time complexity and space complexity**: This is possible because of hybrid data structures. To achieve a balance based on the unique requirements of the application, data structures with various time and space characteristics can be combined. This makes it possible for developers to tailor their solutions to the limitations of the current challenge.

**Flexibility and adaptability**: Hybrid data structures allow for dynamic switching between several underlying data structures in response to shifting situations or demands. They are highly suited for situations where the workload or data distribution may change over time due to their versatility, which enables effective modifications as needed.

**Added features**: Hybrid data structures can offer extra features that may not be present in a single data structure. Combining various data structures enables the achievement of functions that are difficult to perform when using a single data structure alone. When dealing with specialized requirements or complex data processing scenarios, this can be quite helpful.

## Objectives of this project:

The main objective of this project is to teach us the benefits of the hybrid data structure and how useful it would be in various practical applications. We recently completed the project of free RTOS where we saw that the hybrid data structures can be used to simplify our work.  For example, we saw that the ready list in the project which is probably one of the most important components was implemented by using an array of doubly linked lists. What this did was it made the implementation of the project easier. Here are some more practical applications:

**Tries with Hash Tables**: Tries are efficient data structures for storing and searching strings. By combining tries with hash tables, we can achieve a hybrid data structure that provides the best of both worlds. The try data structures handles prefix-based searches and supports efficient insertion and deletion of keys, while the hash table improves the search performance for longer keys. This hybrid structure is commonly used in systems that require fast string lookups, such as spell checkers, autocomplete systems, and IP routing tables.

**Skip Lists with Binary Search Trees**: Skip lists are probabilistic data structures that provide quick search, insertion, and deletion operations with anticipated O(log n) time complexity. Skip lists with binary search trees. Skip lists and binary search trees (BSTs) can be combined to form a hybrid structure that benefits from the simplicity and skip pointers of skip lists as well as the BST's robust worst-case time complexity guarantees. This hybrid structure is frequently used in balanced search tree implementations because it performs better and is more straightforward than more conventional self-balancing tree designs like AVL trees or Red-Black trees.

**Radix Trees with Hash Tables**: Radix trees (also known as Patricia tries) are compact data structures used for efficient storage and retrieval of key-value pairs. By combining radix trees with hash tables, we can create a hybrid data structure that leverages the radix tree's space efficiency and prefix-based search capabilities, while the hash table improves the search performance for longer keys or values. This hybrid structure is commonly used in systems that require efficient key-value storage and retrieval, such as file systems, databases, and network routing tables.

**Fusion Trees**: Fusion trees are hybrid data structures that combine the advantages of different tree structures to achieve efficient search and update operations.

To combine the benefits of several data structures for the best performance, memory efficiency, or functionality, hybrid data structures are chosen and designed according to the needs and characteristics of the application domain.

## Overview of the Hybrid Data Structure

The hybrid data structure which we have chosen is a radix heap which is an array of lists where every list is a different radix position. A radix heap is essentially a data structure that combines the properties of a binary tree and a radix tree. It is primarily used to efficiently maintain a collection of non-negative integers, supporting efficient operations such as insertion, deletion, and finding the minimum element.

A radix heap is a specialized data structure used for maintaining a collection of elements with associated non-negative integer keys. It works especially well in situations where there are few and wide-ranging keys. Radix heap is appropriate for applications that call for effective priority queue operations since it provides quick insertion and extraction of the minimal element.

Based on their binary representations, the radix heap groups elements into various radix buckets. From the least significant bit (LSB) to the most significant bit (MSB), each bucket represents a distinct radix position. Within each bucket, elements with the same radix position are clustered together.

The fundamental principle of radix heap is to efficiently find the radix position of the minimal element using bitwise operations. Radix heap reduces the need for comparisons or sorting within each bucket by maintaining distinct radix buckets. By locating the non-empty bucket with the smallest radix position and removing the element from it, this enables constant-time extraction of the minimum element.

With time complexity of O (1) for insertions and O (log N) for extract-min operations, where N is the number of elements in the heap, the Radix Heap offers effective operations. When you have a lot of elements and a small number of keys or priorities, it is quite helpful.

Overall, radix heap provides a specialized data structure for efficient handling of sparse integer keys, offering improved performance compared to traditional heap structures in certain scenarios.

For the implementation of radix heap we primarily use an array of lists where each and every index is a based on the binary representation of the element. The radix positions represent the bit positions from the least significant bit (LSB) to the most significant bit (MSB) of the keys.

Each radix position is associated with a bucket in the radix heap. The buckets are typically implemented as arrays or linked lists. The index of the bucket corresponds to the radix position. For example in our project we calculate the index by taking the logarithm of the rightmost set bit of the binary representation of the number. In some cases the index is calculated by taking the other methodologies. In another case the bucket at index 0 represents the least significant bit (LSB), the bucket at index 1 represents the next bit, and so on. The elements with the same radix position are stored in each bucket. Depending on the implementation, each bucket may utilise a different data structure. Any appropriate data structure that enables effective insertion and removal is acceptable, including a straightforward array, a linked list, and others.

Radix heap provides quick access to components for operations like insertion, extracting the smallest element, and merging buckets while performing operations by classifying items into buckets based on their radix positions.

## Advantages of Radix Heap:

**A radix heap has the following benefits:**

◊ Radix heap offers an effective implementation of a priority queue, making it possible to insert and remove the minimal element quickly. Depending on the implementation, the time complexity for these operations is O(1) or O(log n), making them acceptable for applications that call for frequent updates and retrievals of the smallest element.

  Radix heaps, which include binary heaps and Fibonacci heaps, can be more space-efficient than conventional priority queue data structures. It does not require additional data structures like parent pointers or auxiliary arrays since it employs a compact representation based on radix positions and bit patterns.

◊ **Fast Merge Operation**: During some operations, Radix heap provides effective bucket merging. When many priority queues need to be joined or constantly updated, this can be helpful. Performance is enhanced since merging is typically quicker than merging different priority queues.

  Radix heap can be configured for distributed systems and parallel computing. It has a built-in structure and indexing that permits concurrent operations on various buckets or radix positions, enabling the parallelization of some operations and possibly enhancing processing speed overall.

◊ **Suitable for Specialized Applications**: Radix heap is particularly useful in specific applications that involve large numbers, such as computational geometry, graph algorithms, and network routing. It is well-suited for scenarios where the keys have a known bit length or a limited range, as it takes advantage of the binary representation of keys to optimize operations.

Overall, the advantages of using a radix heap include efficient priority queue operations, space efficiency, fast merging capabilities, suitability for parallel processing, and applicability to specialized domains. To decide if a radix heap is the best option, it's crucial to consider the specifics of the program and its requirements.

**The primary motivations for using a hybrid data structure are as follows:**

**Performance that is Tailored**: By integrating the advantages of several data structures, hybrid data structures are created. This results in a structure that is particularly optimized for tasks or problem domains. A hybrid data structure can deliver performance that is adapted to the needs of the current challenge by utilizing the distinctive qualities of many data structures.

**Trade-off optimization**: Each data structure has a unique set of advantages and disadvantages. It is feasible to optimize the trade-offs between different criteria, such as time complexity, space complexity, and ease of implementation, by integrating them in a hybrid structure.

**Customized Data Access**: Depending on the problem being handled, hybrid data structures may be created to offer customized data access patterns. Combining one data structure's indexing capabilities with another's effective traversal algorithms can achieve this.

**Handling Complex Relationships**: Some issues include intricate connections between data pieces that are difficult for a single data structure to adequately represent. To manage these relationships more efficiently, hybrid data structures might integrate several data structures.

**Adapting to Dynamic Scenarios**: Hybrid data structures can be designed to adapt to dynamic scenarios where the characteristics of the data or the workload change over time. By combining different data structures, the hybrid structure can switch between them based on the current state or conditions. This adaptability allows for efficient handling of varying scenarios and ensures optimal performance in dynamic environments.

**Application-Specific Optimization**: Hybrid data structures are often developed to address specific challenges or limitations in a particular application domain. For example, in database systems, hybrid indexing structures combine the benefits of different index types to optimize query performance for various data access patterns. Similarly, in computational geometry, hybrid data structures combine spatial partitioning techniques with geometric primitives to efficiently handle geometric queries and operations.

Just how Hydrogen and Oxygen combine to give Water which is a compound with completely different properties compared to the elements such as water is used to put off fire whereas hydrogen and Oxygen act as a fuel similarly when we combine two or more data structures, we get certain properties in the new hybrid which make it far more useful compared to the base data structures.

## *Implementation of the Hybrid Data Structure*

**GitHub Repository Link:** https://github.com/FirefoxSRV/Algoxperts/tree/main

As we have mentioned previously the data structure which we have chosen is a radix heap which we have implemented in both python and C++. This is to make sure we can analyze how the data structure has improved the execution time in an interpreter as well as a compiler. The radix heap is a data structure which can be implemented by either as an array of linked lists or as a normal 2D array where each index is calculated by using the binary representation of each and every number given as an input. The purpose of a radix heap as discussed before is to efficiently store and retrieve the key – value pairs, with a focus on fast minimum extraction. It is particularly useful in scenarios where operations like insertion and extraction of minimum values are frequent and performance critical. The radix heap offers several advantages over traditional data structures like binary heap or balanced search trees such as efficient minimum extraction, time complexity, space complexity, key specific operations and above all it is a hybrid data structure. The purpose of a radix heap is to provide a specialized data structure that optimizes the handling of key-value pairs with binary representations, especially when the focus is on efficient minimum extraction operations.

To implement our code in python we have implemented in two methods. In one method we have used a bucket array/2d array and in another method we have used an array of linked lists. So in both the methods we will get a detailed analysis of how radix heap is better compared to a normal array and a linked list . We will get to see that using a radix heap will help us simplify complex problems with effortless ease as it reduces the time complexity and space complexity drastically. In large problems the complexity of each and every data structure we use becomes important as minor differences will cause huge effects.

## Implementation of radix heap as a bucket array / 2d array

What each component does?

Buckets: The buckets in a radix heap are responsible for storing the elements based on their radix positions. Each bucket represents a specific radix position, ranging from 0 to the maximum radix position (in this case, 63 for 64-bit integers). The elements are stored in the buckets in an unordered manner. When inserting a new value, it is placed in the corresponding bucket based on its radix position.

Minimum Value Tracker: The minimum value tracker is a variable that keeps track of the current minimum value in the radix heap. It starts with a value of None when the radix heap is empty. When a new value is inserted, the minimum value tracker is updated if the inserted value is smaller than the current minimum value or if the heap is empty. This ensures that the minimum value is always accurate and up to date.

Finding the Minimum Value: To extract the minimum value from the radix heap, the code iterates over the buckets to find the bucket with the smallest radix position that contains elements. This is done by iterating from the lowest radix position to the highest and checking if each bucket is empty or not. Once the bucket with the smallest radix position is found, the minimum value is extracted by popping the first element from that bucket. If the bucket becomes empty after the extraction, the minimum value is updated by finding the next minimum value from the remaining non-empty buckets.

Updating the Minimum Value: After an extraction, if the bucket from which the minimum value was extracted becomes empty, the code iterates over the remaining buckets to find the next minimum value. This is done by comparing the first element of each non-empty bucket and updating the minimum value accordingly. By iterating only over the non-empty buckets, the algorithm avoids unnecessary comparisons and ensures efficiency in finding the next minimum value.

How to approach the coding part step by step:

- ◊ In this code first the radix heap class is declared and the radix heap data structure is declared as an array of lists where each list represents a bucket for a specific index position. The number of buckets is based on the number of bits in the data type. Simultaneously a variable to track the minimum value is declared, this is because the most important function of the radix heap is to extract out the minimum value.
- ◊ Then we declare an is_empty() function to check if the radix heap has some element or not.

◊ Then we declare an insert() function to insert the element into the required index after calculating it based upon its radix value. This is the most important part as this calculation determines where the element should be inserted. This is done in different ways in various codes. In our code we first calculate the binary representation find the position of the rightmost set bit(bit which has its value as 1) and take the logarithm of that value and insert the element in that particular index. The important thing here is that there is no sorting between the elements being inserted in a particular index to maintain the average time complexity of O(1).To perform this calculation the math library is imported.

◊ The extract_min method finds the bucket with the minimum value by iterating through the buckets and returns the minimum value by popping it from the bucket. If the bucket becomes empty after the extraction, the minimum value is updated.

◊ The find_min_bucket method iterates through the buckets and returns the index of the first non-empty bucket.

◊ The update_min_value method iterates through the buckets, finds the minimum value among the non-empty buckets, and updates the min_value variable.

◊ The deletion operation is used to remove the particular element from the bucket

◊ The print_heap method prints the content of each non-empty bucket.

The interplay between the buckets and the minimum value tracker ensures that the minimum value is always maintained correctly and can be efficiently extracted from the radix heap. The buckets store the elements based on their radix positions, allowing for efficient insertion and extraction operations. The minimum value tracker keeps track of the current minimum value, enabling quick access to the minimum element in the heap.

## Trade-offs:

◊ The biggest trade off in the radix heap implementation is space complexity/ memory usage for the buckets which is an array of lists. It allocates 64 buckets to accommodate 64 bit integers. This fixed number of buckets regardless of the number of elements being inserted can result in unused memory.

◊ The code uses a linear search approach to find the bucket with the minimum radix position that contains elements. It iterates through all the buckets, regardless of whether they are empty or not, until it finds the first non-empty bucket. This linear search can become inefficient if there are many empty buckets, as it introduces unnecessary iterations

## *Implementation of radix heap as an array of linked lists*

The data structure Radix Heap is implemented as a combination of Array and Queue data structures, where queue is implemented using Linked List. An array of queues is implemented. The array's size has been decided by the maximum number of bits used to represent the elements in the heap. The index where a particular element is stored is decided by the least bit position where a set bit, 1, is found. For instance, $12_{10} = 1100_2$, has its least bit position of 1 at index 2 when indexing from 0. Therefore, the element 12 is enqueued in the queue which is present in the 2nd index of the array. The index of the array where the element's queue is enclosed represents the priority of the queue elements. Lower the index value, higher the priority.

### Implementation in Python

Classes for representing node, queue, and heap – **Node**, **Queue**, **Radix Heap** – are used.

Node class is used to store node data – node.data and the pointer to the next node – node.next.

```
class Node:

    def __init__(self, value):

        self.data = value

        self.next = None
```

Queue class is used to store nodes in a linked list with insertion and deletion pertaining to queue data structure. A head pointer marks the starting node of the linked list. Additionally, a tail pointer is used to simplify the insertion operation by reducing the time complexity to constant value, O(1).

```
class Queue:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    def push(self, element):
        node = Node(element)
        if self.head is None:
            self.head = node
            self.tail = node

        else:
            self.tail.next = node
            self.tail = node

        self.size += 1
```

The insertion operation is designed to not depend on the number of elements in the linked list by making use of the tail pointer which keeps track of the element at the end. As far as queue structure is concerned, insertion takes place in the end of the list, leaving us with constant time complexity by equipping a tail pointer.

RadixHeap class is the main class representing the data structure radix heap. This consists of bucket array, each holding a linked list for the queues. Assuming max. bits required for representing heap elements to be 32, array buckets is initialised with 32 indices containing 32 queues – one on each index. Insertion of elements is done after calculation of index value. This is done by bitwise AND operation between the element value and its corresponding 2's complement, followed by logarithmic value computation to the base 2, i.e.,

$\text{int}(\log_2(\text{value} \,\&\, \text{-value}))$ yields the index value for the element.

```
class RadixHeap:
  def __init__(self):
    self.buckets = [Queue() for _ in range(32)]
    self.min_value = None


  def insert(self, value):
    self.buckets[int(math.log2(value & -value))].push(value)
    if self.min_value is None or value < self.min_value:
      self.min_value = value
```

In RadixHeap class, a function is implemented to find the minimum element which is the element with highest priority while ensuring queue functioning. Scanning through the indices of the bucket array, 1st element in the non-empty queue is returned which is the element with highest priority.

```
def min_element(self):
    for cont in self.buckets:
      if cont.size > 0:
        return cont.head.data
```

RadixHeap.update_min_value() function is used to update the minimum value among the elements in the heap. This is done by traversing through the entire heap, bucket-wise, then traversing corresponding linked list, to compare and set the minimum value in the heap.

```
def update_min_value(self):
    for cont in self.buckets:
      if cont.size > 0:
        curr = cont.head
        min_ = self.min_value
        while curr:
          if curr.data < min_:
            self.min_value = curr.data
          curr = curr.next
```

RadixHeap.delete() function takes in an element value as parameter and deletes the element from the queue in the corresponding bucket array. Deleting an element in a linked list is done by searching for the element and changing the next pointer of its adjacent nodes. The size of the linked list (queue) is then decremented.

```
RadixHeap.delete()
def delete(self, value):
    self.buckets[int(math.log2(value & -value))].delete(value)
    self.min_value = self.min_element()
    self.update_min_value()
```

```
Queue.delete()
```

```
def delete(self, element):

    curr = self.head

    if curr.data == element:

      self.head = self.head.next

    while curr.next:

      if curr.next.data == element:

        curr.next = curr.next.next

        self.size -= 1

        if self.tail.data == element:

          self.tail = curr

        flag = 1


        break

      curr = curr.next

    if flag != 1:

      print("Element not found!")
```

RadixHeap.extract_min_element() is used to remove the element which is in the front position of the queue with the highest priority. This is done by traversing the bucket array for the first non-empty queue, followed by removal of the element pointed by the head pointer. This fn. calls Queue.pop() for the linked list operation on removing the head element.

```
RadixHeap.extract_min_element()

def extract_min_element(self):

    for cont in self.buckets:

      if cont.size > 0:

        cont.pop()
```

```
            return
```

```
Queue.pop()

def pop(self):

    min_element_return = self.head

    self.head = self.head.next

    print("Element removed: ", min_element_return.data)
```

RadixHeap.heap_merge() function is used to merge two radix heaps. The corresponding queues in each bucket array of both the heaps are merged using linked list operations involving assignment of head and tail pointers. The size of both the queues ( which are to be merged ) are considered and appropriate merging operations are implemented.

```
def heap_merge(self, other):

    for i in range(len(self.buckets)):

      if self.buckets[i].size>0 and other.buckets[i].size>0:

        self.buckets[i].tail.next = other.buckets[i].head

        self.buckets[i].tail = other.buckets[i].tail


      else:

        if self.buckets[i].size==0 and self.buckets[i].size>0:

          self.buckets[i].head = other.buckets[i].head

          self.buckets[i].tail = other.buckets[i].tail


        if self.buckets[i].size>0 and self.buckets[i].size==0:

          pass
```

Integration and Interplay of Data Structures

The data structures array and queues (linked list) work together to implement a queue with its "first in, first out" (FIFO) property while accommodating priorities for elements. The buckets in the array sort the elements by priority, and the linked list allows for insertion and deletion of elements with constant time complexity. Together, they create a hybrid structure that provides a superior implementation of priority queues, combining efficiency with prioritization.

## Trade-Offs in the Implementation

- ◊ Radix heap is implemented assuming to accommodate all integers which are represented by a maximum of 32 bits. This results in creating 32 bucket arrays, leading to wastage of memory. Creating a fixed number of buckets can result in higher space complexity.

- ◊ A linear iteration is equipped in finding the first non-empty bucket, i.e., finding the first array index with non-empty queue. The case with many empty buckets in the higher priority indices shows the inefficiency of this search method.

## *Practical Details*

### *Real time application (Traffic control system): -*

The objective of a traffic signal control system is to effectively regulate intersection traffic flow. Traditional traffic signal systems run on fixed timings or set schedules, which might not be able to adjust well to changing traffic conditions. By dynamically altering signal timings in response to real-time traffic data, intelligent traffic signal control systems seek to overcome this constraint.

When choosing the best traffic light layout for each intersection, radix heaps are taken into consideration. A traffic signal configuration, as used in this context, describes the precise timing and order of green, yellow, and red lights for various lanes at an intersection. Every configuration is given a non-negative integer key that measures the amount of time that cars must wait at that intersection.

→Each traffic light arrangement is given a key based on the amount of time that vehicles would have to wait at the intersection. Numerous variables, like traffic volume, congestion levels, historical data, or real-time sensors, can be used to compute this waiting time.

→Building the Radix Heap: The assigned keys are used to build the radix heap data structure. A radix heap is a particular kind of priority queue that arranges items according to their keys. It enables effective element retrieval with the smallest key value.

→Traffic Signal Optimization: To choose the traffic signal configuration with the shortest waiting time, the radix heap is employed. The system finds the best signal timings for each intersection in real time by continually extracting the configuration with the lowest key from the radix heap.

→Dynamic Adjustment: The system continuously modifies the waiting time values and redistributes keys to the traffic signal configurations as traffic conditions change. This makes sure that the radix heap always reflects the volume of traffic in use and makes it possible to choose the configurations that will result in the fewest waiting times.

→Improved Traffic Flow: The intelligent traffic signal control system aims to optimize traffic flow, reduce congestion, and minimize waiting times at intersections by dynamically adjusting the traffic signal timings based on real-time data and using the radix heap to effectively select the optimal configurations.

ALGORITHM:

1. Initialize:
   - Create a radix heap data structure.
   - Assign initial keys to each traffic signal configuration based on estimated waiting times.
2. Traffic Monitoring:
   - Continuously monitor traffic conditions using sensors, cameras, or other data sources.
   - Update the waiting time values for each traffic signal configuration based on the current traffic data.
3. Key Update:
   - Recalculate the keys for each traffic signal configuration based on the updated waiting time values.
4. Radix Heap Operations:
   - Insert each traffic signal configuration into the radix heap, using its key as the priority.
   - Heapify the radix heap to maintain the heap property.
5. Signal Selection and Timing Adjustment:
   - While there are configurations in the radix heap:
   - Extract the configuration with the minimum key from the radix heap.
   - Apply the selected configuration's signal timings to the corresponding intersection.
   - Monitor the effectiveness of the selected configuration and gather feedback.
6. Dynamic Key Update:
   - Based on the feedback received, update the waiting time values and recalculate the keys for the used and neighboring configurations.
   - Update the keys in the radix heap accordingly.
7. Repeat Steps 4-6 until the desired optimization criteria are met or a termination condition is satisfied.

FEW MORE APPLICATIONS: -

1. Navigation Systems: Radix heaps can be used in navigation systems to determine the shortest path between two points. The algorithm must choose the most practical route through a network of roads based on distances or journey times. By allocating non-negative integer keys to each potential path, radix heaps may handle the available routes effectively. The distances or journey periods are represented by these keys. The navigation system may then quickly choose the route with the minimum key, which equates to the lowest distance or trip time, thanks to the radix heap method. The technology can enhance the overall travel experience and offer consumers the most effective routes by choosing the next destination to navigate too effectively.
2. Network Design: The objective of network design or infrastructure planning is to connect various places while keeping costs to a minimum and guaranteeing effective network coverage. Radix heaps, which use Prim's algorithm, can help with this procedure. A minimum spanning tree in a graph, where each vertex denotes a location and each edge denotes a potential connection, is found using Prim's approach. At each stage of extending the tree, it chooses the edge with the least amount of weight. It effectively manages the connections that are already in place and aids in choosing the next edge with the least amount of weight, ensuring that the minimal spanning tree is built in the best possible way. By linking the locations with the lowest overall cost while obtaining the desired results, these aids network planners in building cost-effective and efficient networks.
3. File Compression: In file compression methods like Huffman coding, radix heaps can be used. By allocating variable-length codes to characters based on their frequency, the Huffman coding technique allows for the compression of data. In order to build the Huffman tree, which is needed to create the codes, radix heaps are used. During the creation of the tree, it effectively chooses the two nodes with the two lowest frequencies. The procedure effectively merges the nodes with the lowest frequencies to produce the Huffman tree by arranging the characters according to their frequencies using a radix heap. This makes it possible to create codes for frequently occurring characters that are shorter, leading to more effective file compression. As a result, the compressed files require less storage space and move over networks more quickly or store on devices with limited capacity.
4. Real-time Systems: In real-time systems, where activities or events must be handled in accordance with set deadlines or timestamps, radix heaps are useful. Effective scheduling is essential for real-time systems because they demand timely completion of activities or occurrences. Radix heaps can help with this by

managing a group of jobs or events with non-negative timestamps effectively. The system may swiftly get the subsequent task or event with the shortest timestamp thanks to the radix heap method. By doing this, you can make sure that actions or events are carried out in the right order, on time, or in the intended order. Real-time systems can ensure accurate timing and responsiveness by effectively managing and prioritizing tasks or events.

5. Resource Allocation: Resource allocation scenarios can use radix heaps, especially in cloud computing environments. Multiple jobs or processes may compete for scarce resources in such contexts. By ranking activities according to their urgency or relevance, radix heaps can help with resource allocation. Each job or process can be given a non-negative integer key reflecting its priority or urgency by the system. The radix heap makes it simple to choose the job or process with the greatest priority, guaranteeing that the most important tasks are given priority when allocating resources. Resource allocation can be optimized by effective task management and prioritization, resulting in higher system utilization and performance overall.
6. Financial Systems and Databases: In financial systems and databases, radix heaps offer efficient data retrieval and analytical capabilities. They make it possible to organize time-based data effectively and retrieve it when needed. They also make it possible to analyze both historical and real-time data. Financial systems and databases benefit from the effective processing of data entries with non-negative integer keys using radix heaps in terms of efficiency, responsiveness, and decision-making.

## KEY DIFFERENCES BETWEEN (2-D ARRAY AND LINKED LISTS):-

In the context of a traffic light project, let's discuss which code implementation would be better:

Code 1: Advantages for Traffic Light Project:

8. Customizable Configuration: Code 1 allows associating specific configurations with keys. This feature can be useful in a traffic light project where different configurations correspond to different traffic scenarios, such as peak hours, low traffic, or special events. It provides flexibility in defining and managing traffic configurations.

Disadvantages for Traffic Light Project:

9. Inefficient Extract Minimum Operation: The extract_min() operation in Code 1 has a linear time complexity of O(n), where n is the number of configurations. In a traffic light project with a large number of configurations, extracting the minimum configuration from the heap can become slow and inefficient. This could impact real-time decision-making for signal selection and timing adjustments.

Code 2: Advantages for Traffic Light Project:

10. Efficient Extract Minimum Operation: Code 2 uses a bucket-based approach with linked lists, resulting in a constant time complexity of O(1) for extracting the minimum configuration. In a traffic light project with a large number of configurations, this approach ensures fast and efficient selection of the next configuration to be applied.
11. Linked List Structure: Code 2's linked list structure allows for efficient insertion and deletion operations. This can be beneficial when updating waiting time values and recalculating keys based on feedback from the traffic intersection.

Disadvantages for Traffic Light Project:

12. Limited Key Range: Code 2 uses a fixed-size list of buckets to store the configurations, limiting the range of possible keys. In a traffic light project with a diverse set of configurations, this limitation may not provide enough flexibility to cover all possible traffic scenarios.

Considering the specific needs of a traffic light project, Code 2 is generally better suited due to its efficient extract_min () operation and linked list structure. The ability to quickly select the next configuration and update waiting times based on real-time feedback is crucial in traffic management systems. However, if the project requires extensive customization and a wider key range, Code 1's customizable configuration approach may be more suitable

## ***PERFORMANCE ANALYSIS***

### Analysis of functions and Time Complexities - 2D Array

```
RadixHeap() : min_value(-1) {}
```

The class constructor's time complexity is just O(1). It initializes the min_value to -1, which means there are no contents in the radix heap right now.

```
bool is_empty() {
   return min_value == -1;
}
```

The function is_empty() also has the time complexity of O(1) as it just checks if the radixHeap is empty or not. If its empty, min_value == -1 will return true.

```
void insert(int value) {
      int mask = value & -value;
      int index = static_cast<int>(log2(mask));
      buckets[index].push_back(value);

      if (min_value == -1 || value < min_value) {
         min_value = value;
      }
}
```

All other functions are of O(1) except the log2(mask). Here, we are calculating the logarithmic value of mask in line 3 to find the index of the value to be inserted. The Big-Oh time complexity of log2(mask) is O(logN) , where N is the maximum value in the heap.

```
int extract_min() {
      int min_bucket = find_min_bucket();
      int min_value = buckets[min_bucket].front();
      buckets[min_bucket].pop_front();

      if (buckets[min_bucket].empty()) {
         update_min_value();
      }
      return min_value;
  }
```

The above extract_min() funtion has a time complexity of O(logN).

- In line2, the find_min_bucket() returns the minimum element in the radixHeap. This is O(1)
- In line 3, buckets[min_bucket].front() and line 4's popfront() also takes O(1).
- The update_min_value() takes O(logN) [ where N is the max value of the radixheap ]as it involves searching through the elements and updating the min_value.

So, as a whole, the worst case scenario adds up to O(logN).

```
int find_min_bucket() {
      for (size_t i = 0; i < 64; i++) {
         if (!buckets[i].empty()) {
            return static_cast<int>(i);
         }
      }
      return -1; // No non-empty bucket found
  }
```

The find_min_bucket() function has a time complexity of O(64) as it returns the first non-empty bucket. The worst case scenario is when all the buckets are empty except the last one. So, it takes the 64th bucket as its min_bucket. So, the number of elements here is 64, that is N. So, the complexity is O(N).

```
void update_min_value() {
    for (size_t i = 0; i < 64; i++) {
       if (!buckets[i].empty()) {
           min_value=*std::min_element(buckets[i].begin(),buckets[i].end());
            return;
         }
      }
      min_value = -1; // No non-empty bucket found
  }
```

The update_min_value() function takes a time complexity of O(N). For each an every iteration, it finds the min_element of the corresponding bucket. If the new value is less than the existing minimum value, then the new value is set as the new minumum.

Since there are 64 buckets in the RadixHeap implementation, the overall worst-case time complexity of update_min_value() can be expressed as O(64 * N), which simplifies to O(N).

Even in average case, a bucket contains an average of N/64 elements. So, O(64 * N/64 ) = O(N) is its time complexity. Finally, the time complexity of this function is O(N) in both worst and average case.

```
int get_min() {
    return min_value;
}
```

getmin() is another function that has only O(1) complexity. It just returns the minimum value.

```
size_t size() {
    size_t count = 0;
    for (const auto& cont : buckets) {
      count += cont.size();
    }
    return count;
}
```

The time complexity of this size () function is O(N). The function iterates over buckets in the buckets array and sums up the sizes of each bucket. The cont.size() is a function of O(1) complexity. Hence, it computes the count in every iteration of bucket and returns the total size. Hence, as a whole the time complexity is O(N).

```
void clear() {
    for (auto& cont : buckets) {
      cont.clear();
    }
    min_value = -1;
}
```

The function iterates over all the buckets in the buckets array and calls the clear() function on each bucket. So, the cont.clear() is of O(1) complexity. On calling it multiple times, (64 to be exact) the complexity of this is O(64). That is O(N) because it calls the function as many times as the number of elements in the cont.

```
void merge(RadixHeap& other_heap) {
    for (int i = 0; i < NUM_BUCKETS; i++) {
      int other_size = other_heap.bucket_sizes[i];
      for (int j = 0; j < other_size; j++) {
        buckets[i][bucket_sizes[i] + j] = other_heap.buckets[i][j];
      }
      bucket_sizes[i] += other_size;
      other_heap.bucket_sizes[i] = 0;
    }
    update_min_value();
    other_heap.update_min_value();
  }
```

The function iterates over each bucket (a constant number of buckets, NUM_BUCKETS = 64) and performs a simple array copy for each non-empty bucket of the other heap. The time complexity of the merge() function is O(M), where M is the total number of values in the other heap.

```
void delete_min() {
    int min_bucket = find_min_bucket();
    for (int i = 1; i < bucket_sizes[min_bucket]; i++) {
      buckets[min_bucket][i - 1] = buckets[min_bucket][i];
    }
    bucket_sizes[min_bucket]--;

    if (bucket_sizes[min_bucket] == 0) {
      update_min_value();
    }
  }
```

The function finds the minimum value by searching for the non-empty bucket with the minimum value, which takes O(NUM_BUCKETS) = O(64) time in the worst case. That is O(N) as in worst case complexity.

Time complexity comparisons with arrays

1. Insertion:
    - Normal Array: In a normal array, insertion involves finding the appropriate position and shifting elements to make space for the new element. This operation has a time complexity of O(n) in the worst case, as it requires shifting elements and resizing the array if necessary.
    - RadixHeap: The insertion operation in RadixHeap involves finding the radix position of the value and inserting it into the appropriate bucket in a sorted manner. The time complexity of the insertion operation in RadixHeap is determined by the number of nodes in the target bucket, which can be approximated as O(n/64), where n is the number of nodes in the heap. In practice, it is usually faster than O(n) due to the limited number of nodes per bucket.
2. Deletion (Extract Min):

   - Normal Array: In a normal array, deleting the minimum element requires searching for the minimum value, which has a time complexity of O(n) in the worst case. After finding the minimum, the element is removed by shifting the subsequent elements to fill the gap, which also has a time complexity of O(n).
   - RadixHeap: In RadixHeap, the extract_min operation involves finding the minimum value by iterating over the buckets in O(1) time and deleting the minimum node in O(1) time. If the bucket becomes empty, updating the minimum value takes O(n) time in the worst case. Overall, the time complexity of extract_min in RadixHeap is O(1), with the update_min_value operation having a separate time complexity of O(n).
3. Searching:
   - Normal Array: Searching for an element in a normal array requires iterating over the array and comparing each element to the target value. In the worst case, this operation has a time complexity of O(n) as all elements need to be checked.
   - RadixHeap: RadixHeap is not designed for efficient searching. It is primarily optimized for insertions and extractions of the minimum value. Searching for a specific value in RadixHeap would require iterating over all the buckets and checking each node, resulting in a time complexity of O(n).

## Analysis of functions and Time Complexities - LL and array

```
class RadixHeapNode {
public:
   int value;
   RadixHeapNode* next;

   RadixHeapNode(int value) : value(value), next(nullptr) {}
};
```

The time complexity of the RadixHeapNode constructor is O(1). This is because the constructor initializes the value member variable with the provided value and sets the next pointer to nullptr.

```
RadixHeap() : buckets(new RadixHeapNode*[64]{}), min_value(-1) {}
```

This is the constructor of the RadixHeap class. The time complexity of the RadixHeap constructor is O(1). The buckets array is initialized with a size of 64 and all elements are set to nullptr using the empty brace initialization {}. The min_value is initialized to -1.

```
~RadixHeap() {
      for (int i = 0; i < 64; i++) {
         RadixHeapNode* node = buckets[i];
         while (node != nullptr) {
            RadixHeapNode* temp = node;
            node = node->next;
            delete temp;
         }
      }
      delete[] buckets;
   }
```

The time complexity of the ~RadixHeap() destructor is O(n), where n is the total number of nodes in the radix heap. The destructor iterates over each bucket in the buckets array, which has a constant size of 64. For each bucket, it traverses the linked list of nodes starting from the head node (buckets[i]) and deletes each node.

```
bool is_empty() {
      return min_value == -1;
}
```

This returns the min_value. This is again O(1).

```
void insert(int value) {
      RadixHeapNode* node = new RadixHeapNode(value);
      int radix_pos = get_radix_position(value);
      if (buckets[radix_pos] == nullptr) {
         buckets[radix_pos] = node;
      } else {
         RadixHeapNode* curr = buckets[radix_pos];
         RadixHeapNode* prev = nullptr;
         while (curr != nullptr && curr->value <= value) {
            prev = curr;
            curr = curr->next;
         }
         if (prev == nullptr) {
            node->next = curr;
            buckets[radix_pos] = node;
         } else {
            prev->next = node;
            node->next = curr;
```

```
        }
      }
      if (min_value == -1 || value < min_value) {
        min_value = value;
      }
    }
```

The time complexity of the insert function in the RadixHeap class depends on two main factors:

1. Finding the radix position , and
2. Inserting the node.

Overall, the time complexity of the insert function is O(n), where n is the number of nodes in the bucket (in the worst case). The get_radix_position() function has worst case O(n) that we will discuss now.

```
int get_radix_position(int value) {
      return static_cast<int>(log2(value & -value));
  }
```

Yes, previously in the insert(), we had used this. The value & -value represents the mask value. The log2(mask) represents the index where the value is to me inserted. So, the log2 calculation takes O(n) complexity.

```
int extract_min() {
      int min_bucket = find_min_bucket();
      RadixHeapNode* min_node = buckets[min_bucket];
      int min_value = min_node->value;
      buckets[min_bucket] = min_node->next;
      delete min_node;
      if (buckets[min_bucket] == nullptr) {
        update_min_value();
      }
      return min_value;
  }
```

The time complexity of the extract_min function in the Radix Heap class depends on two main operations:

1. Finding the minimum bucket - This is a time complexity of O(1) ,
2. Extracting the minimum value - The function extracts the min_value and then updates the min_value using the update_min_value() function. This takes O(n).

```
void update_min_value() {
      for (int i = 0; i < 64; i++) {
         if (buckets[i] != nullptr) {
            min_value = buckets[i]->value;
            return;
         }
      }
      min_value = -1;
}
```

As discussed above, the update_min_value() takes O(N) complexity as in the worst case, it needs to iterate till 64. So, that's N again!

```
void print_heap() {
      for (int i = 0; i < 64; i++) {
         std::cout << "Bucket " << i << ": ";
         RadixHeapNode* node = buckets[i];
         while (node != nullptr) {
            std::cout << node->value << " ";
            node = node->next;
         }
         std::cout << std::endl;
      }
   }
```

In the worst-case scenario, where all the nodes are distributed evenly across the buckets, each bucket would contain approximately n/64 nodes on average, where n is the total number of nodes in the radix heap. Therefore, the time complexity of printing the radix heap would be approximately O(n/64). This can be approximated to be O(n), where n is the number of elements.

Time complexities between Radix Heap - LinkedList-Array and normal array

1. **Insertion**:
   - LinkedList-Array: The insertion operation in the LinkedList-Array implementation has a time complexity of O(1). It simply adds the new value to the appropriate bucket based on its radix position.
   - Normal Array: The insertion operation in the normal array implementation has a time complexity of O(1) on average, assuming a constant number of values per bucket. However, if the bucket becomes full, it needs to shift the existing elements to make space for the new value, which has a worst-case time complexity of O(k), where k is the number of values in a bucket.
2. **Extraction (extract_min):**

- LinkedList-Array: The extraction operation in the LinkedList-Array implementation has a time complexity of O(1). It retrieves the minimum value from the first non-empty bucket and updates the heap accordingly.
- Normal Array: The extraction operation in the normal array implementation has a time complexity of O(k), where k is the number of values in the first non-empty bucket. It needs to shift the remaining values in the bucket to fill the gap left by the extracted minimum value.

3. **Deletion (delete_min):**
    - LinkedList-Array: The delete_min operation in the LinkedList-Array implementation has a time complexity of O(1) similar to extraction. It removes the minimum value from the first non-empty bucket and updates the heap.
    - Normal Array: The delete_min operation in the normal array implementation has a time complexity of O(k), where k is the number of values in the first non-empty bucket. It needs to shift the remaining values in the bucket to fill the gap left by the deleted minimum value.
4. **Merge:**
    - LinkedList-Array: The merge operation in the LinkedList-Array implementation has a time complexity of O(k), where k is the number of values in the other heap being merged. It iterates over the buckets of the other heap and adds the values to the corresponding buckets in the current heap.
    - Normal Array: The merge operation in the normal array implementation has a time complexity of O(k), where k is the number of values in the other heap being merged. It also iterates over the buckets of the other heap and adds the values to the corresponding buckets in the current heap.

## ***EXPERIMENTAL EVALUATION***

The library used in our code to record the time of each program is the time library. The time library in python provides facilities for measuring time durations and points in time. The library is based on a type-safe representation of time intervals and clocks.

The time module in Python provides various functions and constants for working with time-related operations. Some key components of the time library include:

time.time(): Returns the current time in seconds since the epoch as a floating-point number.

time.sleep(secs): Suspends the execution of the current thread for the specified number of seconds.

time.perf_counter(): Returns the value of a performance counter, which is a clock with the highest available resolution for measuring short durations. It is typically used for measuring code execution time.

time.process_time(): Returns the sum of the system and user CPU time of the current process in seconds.

time.strftime(format[, t]): Converts a struct_time object or a time tuple to a string representation based on the specified format.

time.strptime(string, format): Parses a string representing a time according to the given format and returns a struct_time object.

time.gmtime([secs]): Converts a time in seconds since the epoch to a UTC struct_time object.

time.localtime([secs]): Converts a time in seconds since the epoch to a local struct_time object representing the current time.

time.mktime(t): Converts a struct_time object or a time tuple to a floating-point number representing the time in seconds since the epoch.

```
import time

# Start measuring the execution time

start_time = time.perf_counter()
```

```
        # Your code here

        # Stop measuring the execution time

        end_time = time.perf_counter()

        # Calculate the duration in milliseconds

        duration_ms = (end_time - start_time) * 1000

        # Print the execution time in milliseconds

print("Executiontime:",duration_ms, "milliseconds")
```

This is a sample on how the above time module works.

<u>Time for traffic light system – array of lists:</u>

```
Execution time: 0.34499999310355633 milliseconds
```

<u>Time for traffic light system – linked list and array:</u>

```
Execution time: 0.11719999019987881 milliseconds
```

## *DISCUSSION*

The data structure which we have implemented is a radix heap which has many advantages and disadvantages as we have seen before. A radix heap is conceptually a combination of a radix tree and a binary tree and is implemented as an array of linked lists or an array of lists. We have implemented it in both the models.

### Practicality and Effectiveness:

The radix heap is primarily practical and effective for priority-based operations, its versatility, deterministic behavior, high performance for specific data distributions.

1.Priority Based Operations: Radix heap supports easy extraction and insertion operations with an average case time complexity of O (1) which makes it a highly attractive data structure which can be utilized whenever the elements need to be inserted or extracted based upon their priority. So, it can be used for tasks such as job scheduling in operating systems like RTOS where each job is allocated based upon its priority and within a particular priority the jobs are chosen in a first come first serve manner.

2.Deterministic Behavior: Radix heap ensures deterministic behavior in situations where ordering matters by guaranteeing that elements with the same priority are removed in the order of their insertion.

3.High Performance for specific data distributions: When the data distribution matches the presumption of uniform distribution across the binary digits, radix heap performs especially well. It can perform better in these circumstances than alternative priority queue data structures.

4.Versatility: Radix heap can handle a wide range of data types including integers and fixed size bit sequences. Its versatility makes it applicable in various domains.

### Limitations:

While radix heap has plenty of benefits it has its own limitations as well. Its limitations include memory overhead, sensitivity to data distribution, implementation complexity and limited support for dynamic operations.

(1) Memory Overhead: Radix heaps require memory space for maintaining the buckets corresponding to each bit position. The number of buckets is determined by the number of bits in the fixed-size integers being processed.

(2) Sensitivity to Data Distribution: The radix heap functions best when there is a uniform data distribution. In the case where the data distribution deviates a lot the performance of the radix heap degrades.

(3) Implementation Complexity: Implementation of a radix heap can be complicated because of the intricacies of handling the bucket organization and maintaining the priority order within each bucket.

(4) Limited Support for dynamic operations: Radix heaps is not ideal for dynamic operations like removing arbitrary components or altering priority on the fly. Altering an element's priority or removing any element from a radix heap calls for additional operations that could reduce overall efficiency.

## Potential Future Improvements:

1.Dynamic Update Support: In the future the radix heap can be changed in such a way that it allows us to change the values of the priority dynamically. Once we are able to support dynamic updates it would increase its practicality and widen its range of applications.

2. Performance Optimization: In certain circumstances, such as data distributions with different priority levels, performance may be enhanced by analyzing and optimizing the interactions between the radix heap and linked list components.

3.Customization and Adaptability: By allowing for varied priority hierarchies or the integration of other data structures, the hybrid data structure might be more easily applied in a variety of contexts.

## *CONCLUSION*

In conclusion the radix heap is a hybrid data structure which can be used as a replacement for arrays, linked list and binary trees. It is formed by combining the benefits of a binary tree and a radix sort. Its primary benefit is that it offers insertion and extraction of elements at a time complexity of O (1). The radix heap demonstrates excellent performance when processing fixed size integer data with a uniform distribution. It provides a significant improvement over the traditional binary heaps and linked lists in terms of time complexity for insertion and extraction operations. However, despite its benefits the radix heap does have some limitations such as memory overhead and lack of support for dynamic update. If these problems are fixed in the future the radix heap will eventually become a household name in the field of computer science unlike today where it is not used often by teachers and programmers. The radix heap will have great practical applications and will be used wherever priority is an important factor. Despite these limitations radix heap is still used in areas such as graph algorithms, shortest path algorithms and event driven simulations where an efficient priority queue is necessary. So, in this project we have been able to compare and contrast the radix heap and its individual base data structures in various scenarios by calculating their time for execution. In each and every section in our report we have explained in detail the working of the radix heap, its various operations and their details. We have explained the time complexity of these operations as well and given an explanation of how it is an improvement from their base data structures.

To conclude our project, we are extremely grateful towards our dsa faculty and the institution to provide us with this project of hybrid data structures which has enabled us to learn a lot about the importance and effectiveness of hybrid data structures and how they can be used to enhance the performance compared to normal data structures. Our project Radix Heap has been a great learning curve for us. Thank You

**References:**

https://chat.openai.com/c/4dd28068-3e37-4d2a-97ab-19f5d8f8a003

https://en.wikipedia.org/wiki/Radix_heap

**https://github.com/iwiwi/radix-heap**

**https://www.geeksforgeeks.org/applications-of-heap-data-structure/**