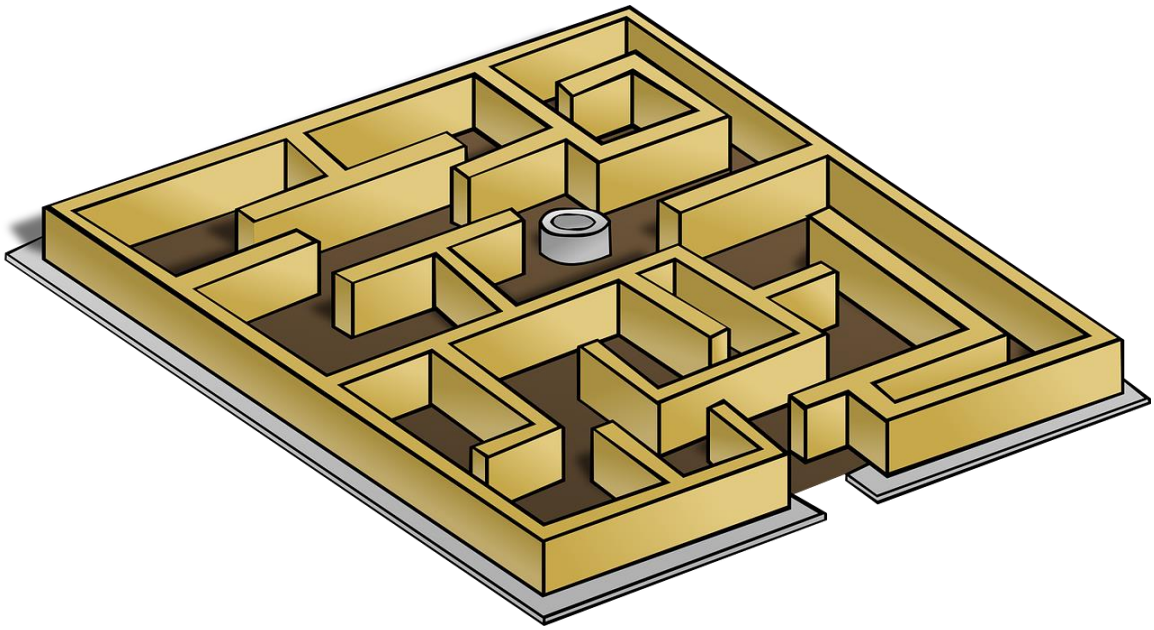


ADVANCED DATA STRUCTURES

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

Interval Trees

An interval tree is a data structure that can be used to represent a set of intervals on a line. Each interval is represented by a pair of numbers, its start and end points. Interval trees can be used to solve a variety of problems, such as finding the intervals that overlap a given point or finding the intervals that are completely contained within another interval.

Interval trees are typically implemented as binary search trees, where each node in the tree represents an interval. The intervals are sorted by their start points, and the children of each node represent the intervals that are completely contained within the interval represented by the node.

Interval trees can be used to solve a variety of problems, such as:

- Finding the intervals that overlap a given point
- Finding the intervals that are completely contained within another interval
- Finding the intervals that are adjacent to each other
- Finding the intervals that are in a given range
- Finding the intervals that are sorted by their start points

Interval trees are a powerful data structure that can be used to solve a variety of problems. They are relatively easy to implement and can be used to solve problems that would be difficult to solve with other data structures.

Example: Let's consider an example of how an interval tree can be used to solve a problem. Suppose we have a set of intervals that represent the times that a group of people are available to meet. We want to find the intervals that overlap so that we can schedule a meeting that will work for everyone.

We can represent the set of intervals as an interval tree. The start points of the intervals are sorted, and the children of each node represent the intervals that are completely contained within the interval represented by the node.

To find the intervals that overlap, we can start at the root of the tree and recursively check each node. If the start point of the current node is less than or equal to the end point of the current node, then the current node represents an interval that overlaps with the current interval. We can then add the current node to the list of overlapping intervals.

We continue recursively checking each node until we reach a leaf node. At this point, we have found all of the intervals that overlap with the current interval.

In this example, we can use an interval tree to find the intervals that overlap so that we can schedule a meeting that will work for everyone.

Implementation

Interval trees can be implemented in any programming language. Here is an example of how an interval tree can be implemented in Java:

```
class IntervalTree {


  private class Node {

    Interval interval;

    Node left;

    Node right;


    Node(Interval interval) {

      this.interval = interval;
```

```
  }
}

private Node root;

public IntervalTree() {
  root = null;
}

public void addInterval(Interval interval) {
  root = addInterval(root, interval);
}

private Node addInterval(Node node, Interval interval) {
  if (node == null) {
    return new Node(interval);
  }

  if (interval.start < node.interval.start) {
    node.left = addInterval(node.left, interval);
  } else {
    node.right = addInterval(node.right, interval);
```

```
    }

    return node;
  }

  public List<Interval> getOverlappingIntervals(Interval interval) {
    List<Interval> overlappingIntervals = new ArrayList<>();
    getOverlappingIntervals(root, overlappingIntervals, interval);
    return overlappingIntervals;
  }

  private void getOverlappingIntervals(Node node, List<Interval> overlappingIntervals, Interval interval) {
    if (node == null) {
      return;
    }

    if (interval.start <= node.interval.end && interval.end >= node.interval.start) {
      overlappingIntervals.add(node.interval);
    }

    getOverlappingIntervals(node.left, overlappingIntervals, interval);
```

```
    getOverlappingIntervals(node.right, overlappingIntervals, interval);
  }
}
```

This is just a simple implementation of an interval tree. There are many other ways to implement interval trees.

Trie

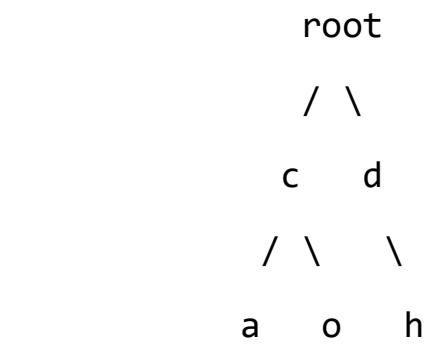
A trie, also known as a prefix tree, is a tree data structure that is used to store a set of strings. Each string is stored as a path from the root of the tree to a leaf node. The nodes in the trie store the characters of the strings, and the edges in the trie store the relationships between the characters.

Tries are a very efficient data structure for storing and searching strings. They can be used to solve a variety of problems, such as:

- String matching: Tries can be used to find all of the strings in a set that match a given pattern.
- Spell checking: Tries can be used to check if a given word is spelled correctly.
- Autocomplete: Tries can be used to suggest words that are likely to be typed next.

Tries are typically implemented as binary trees, where each node in the tree stores a character and the children of each node store the characters that follow the character stored in the node.

Here is an example of a trie that stores the strings "cat", "dog", and "hat":



To find the string "cat" in this trie, we would start at the root node and follow the edges "c" and "a". To find the string "dog", we would start at the root node and follow the edges "d" and "o". To find the string "hat", we would start at the root node and follow the edges "h" and "a".

Tries are a powerful data structure that can be used to solve a variety of problems. They are relatively easy to implement and can be used to solve problems that would be difficult to solve with other data structures.

Implementation

Tries can be implemented in any programming language. Here is an example of how a trie can be implemented in Java:

```
class Trie {


  private class Node {

    char character;

    Node[] children;


    Node(char character) {

      this.character = character;

      children = new Node[26];

    }

  }


  private Node root;


  public Trie() {

    root = null;
```

```
}

public void addString(String string) {
  addString(root, string, 0);
}

private void addString(Node node, String string, int index) {
  if (index == string.length()) {
    return;
  }

  char character = string.charAt(index);
  int childIndex = character - 'a';
  if (node.children[childIndex] == null) {
    node.children[childIndex] = new Node(character);
  }

  addString(node.children[childIndex], string, index + 1);
}

public boolean containsString(String string) {
  return containsString(root, string, 0);
```

```
  }

  private boolean containsString(Node node, String string, int index) {
    if (index == string.length()) {
      return true;
    }

    char character = string.charAt(index);
    int childIndex = character - 'a';
    if (node.children[childIndex] == null) {
      return false;
    }

    return containsString(node.children[childIndex], string, index + 1);
  }

}
```

This is just a simple implementation of a trie. There are many other ways to implement tries.

Suffix Trees

A suffix tree is a data structure that can be used to represent a string. It is a compressed trie of all the suffixes of the string. Suffix trees are used to solve a variety of string-related problems, such as pattern matching, finding distinct substrings in a given string, and finding longest palindromes.

Suffix trees can be constructed using a variety of algorithms. The most common algorithm is Ukkonen's algorithm. Ukkonen's algorithm is a dynamic programming algorithm that constructs the suffix tree one suffix at a time.

Suffix trees can be implemented in any programming language. Here is an example of how a suffix tree can be implemented in Java:

```
class SuffixTree {


  private class Node {

    int start;

    int end;

    Node[] children;


    Node(int start, int end) {

      this.start = start;

      this.end = end;

      children = new Node[26];

    }

  }
```

```
private Node root;

public SuffixTree(String string) {
  root = buildSuffixTree(string);
}

private Node buildSuffixTree(String string) {
  int n = string.length();
  Node root = new Node(0, n - 1);

  for (int i = 0; i < n; i++) {
    addSuffix(root, string, i);
  }

  return root;
}

private void addSuffix(Node node, String string, int index) {
  int c = string.charAt(index) - 'a';
  if (node.children[c] == null) {
    node.children[c] = new Node(index, n - 1);
```

```
  }

  if (index != n - 1) {
    addSuffix(node.children[c], string, index + 1);
  }
}

public boolean containsSuffix(String suffix) {
  return containsSuffix(root, suffix, 0);
}

private boolean containsSuffix(Node node, String suffix, int index) {
  if (node == null) {
    return false;
  }

  if (node.end < index) {
    return false;
  }

  if (node.start > index) {
    return false;
```

```
    }

    if (suffix.length() == 0) {
      return true;
    }

    int c = suffix.charAt(0) - 'a';
    return containsSuffix(node.children[c], suffix, index + 1);
  }

  public List<Integer> getIndexesOfSuffix(String suffix) {
    List<Integer> indexes = new ArrayList<>();
    getIndexesOfSuffix(root, suffix, 0, indexes);
    return indexes;
  }

  private void getIndexesOfSuffix(Node node, String suffix, int index,
List<Integer> indexes) {
    if (node == null) {
      return;
    }

    if (node.end < index) {
```

```
    return;
   }

   if (node.start > index) {
    return;
   }

   if (suffix.length() == 0) {
    indexes.add(node.start);
   }

   int c = suffix.charAt(0) - 'a';
   getIndexesOfSuffix(node.children[c], suffix, index + 1, indexes);
  }

}
```

This is just a simple implementation of a suffix tree. There are many other ways to implement suffix trees.

Bloom Filters

A Bloom filter is a probabilistic data structure that allows for fast and memory-efficient membership testing. It provides a way to answer the question "Is element X in the set?" without storing the entire set of elements. Bloom filters can offer significant space savings compared to traditional data structures like arrays or hash tables.

A Bloom filter consists of a bit array of m bits, where m is a power of two. Initially, all bits are set to 0. To add an element to the Bloom filter, we hash the element k times, where k is the number of hash functions. The hash functions should be chosen so that they are evenly distributed over the range of 0 to m - 1. For each hash function, we set the bit at the corresponding index in the bit array to 1.

To test whether an element is in the Bloom filter, we hash the element k times and check if the corresponding bits in the bit array are set to 1. If all bits are set to 1, then the element is probably in the Bloom filter. However, it is possible for an element to not be in the Bloom filter and still return a positive result. This is called a false positive. The probability of a false positive can be controlled by the number of hash functions used.

Advantages of Bloom Filters

Bloom filters offer several advantages over traditional data structures for membership testing. They are:

- Space-efficient: Bloom filters can offer significant space savings compared to traditional data structures. The space required for a Bloom filter is proportional to the number of bits in the bit array, which is typically much smaller than the number of elements in the set.
- Fast: Membership testing in a Bloom filter is very fast. It takes only O(k) time, where k is the number of hash functions.

- Probabilistic: Bloom filters are probabilistic data structures. This means that there is a small probability that an element will return a positive result even if it is not in the set. This probability can be controlled by the number of hash functions used.

Disadvantages of Bloom Filters

Bloom filters also have some disadvantages. They are:

- False positives: The main disadvantage of Bloom filters is the possibility of false positives. This means that an element may return a positive result even if it is not in the set. This can be a problem if the false positive rate is too high.
- Cannot be used for deletion: Bloom filters cannot be used to delete elements from the set. Once an element is added to the Bloom filter, it cannot be removed.

Applications of Bloom Filters

Bloom filters are used in a variety of applications, including:

- Web filtering: Bloom filters can be used to filter out unwanted content from web pages. For example, they can be used to block pornographic or malicious content.
- Spam filtering: Bloom filters can be used to filter out spam emails. For example, they can be used to block emails that contain certain keywords or phrases.
- Duplicate detection: Bloom filters can be used to detect duplicate files. For example, they can be used to find duplicate images or documents.
- Load balancing: Bloom filters can be used to load balance web servers. For example, they can be used to distribute requests to servers based on their load.

Conclusion

Bloom filters are a powerful tool that can be used for a variety of applications. They offer a good trade-off between space and speed, and they can be used to solve a variety of problems.

Skip Lists

A skip list is a probabilistic data structure that can be used to store a sorted list of elements. It is similar to a linked list, but it has additional levels that allow for faster searching.

The bottom level of a skip list is a regular linked list. The next level up has half as many nodes as the bottom level, and so on. The top level has only one node.

Each node in a skip list has two pointers: a forward pointer and a down pointer. The forward pointer points to the next node in the same level. The down pointer points to the corresponding node in the next level.

To search for an element in a skip list, we start at the top level and follow the forward pointers until we reach a node that contains the element we are looking for. If we don't find the element, we then follow the down pointer to the next level and repeat the process.

Insertion and deletion in a skip list are similar to search. To insert an element, we first find the node where the element should be inserted. We then create a new node with the element and insert it into the list. To delete an element, we find the node that contains the element and then remove it from the list.

Skip lists have a number of advantages over other data structures, such as linked lists and binary search trees. They are typically faster for searching, insertion, and deletion, and they are also more scalable.

Here is an example of how to implement a skip list in Java:

```
class SkipList {


    private static final int MAX_LEVELS = 32;
```

```
    private Node head;

    public SkipList() {
        head = new Node(null, null);
    }

    public void insert(int value) {
        Node current = head;
        Node newNode = new Node(value, null);

        for (int level = MAX_LEVELS - 1; level >= 0; level--) {
            while (current.forward[level] != null && current.forward[level].value < value) {
                current = current.forward[level];
            }
        }

        newNode.down = current;
        for (int level = MAX_LEVELS - 1; level >= 0; level--) {
            newNode.forward[level] = current.forward[level];
            current.forward[level] = newNode;
        }
    }
```

```
    public void delete(int value) {
        Node current = head;

        for (int level = MAX_LEVELS - 1; level >= 0; level--) {
            while (current.forward[level] != null && current.forward[level].value < value) {
                current = current.forward[level];
            }
        }

        if (current.forward[0] != null && current.forward[0].value == value) {
            for (int level = MAX_LEVELS - 1; level >= 0; level--) {
                if (current.forward[level] != null) {
                    current.forward[level] = current.forward[level].forward[level];
                }
            }
        }
    }

    public boolean contains(int value) {
```

```
        Node current = head;

        for (int level = MAX_LEVELS - 1; level >= 0; level--) {
            while (current.forward[level] != null && current.forward[level].value < value) {
                current = current.forward[level];
            }
        }

        return current.forward[0] != null && current.forward[0].value == value;
    }

    private static class Node {

        private int value;
        private Node down;
        private Node[] forward;

        public Node(int value, Node down) {
            this.value = value;
            this.down = down;
            this.forward = new Node[MAX_LEVELS];
```

```
        }
    }
}
```

This is just a simple example of how to implement a skip list in Java. There are many other ways to implement it, and there are also many different ways to optimize it.

Treaps

A treap is a probabilistic data structure that can be used to store a sorted list of elements. It is similar to a binary search tree, but it has a random priority associated with each node. The priority of a node is used to determine the order of the nodes in the tree.

Treaps have a number of advantages over other data structures, such as binary search trees. They are typically faster for searching, insertion, and deletion, and they are also more scalable.

Here is an example of how to implement a treap in Java:

```
class Treap {

   private static final Random random = new Random();

   private Node root;

   public Treap() {
      root = null;
   }

   public void insert(int value) {
      Node newNode = new Node(value, random.nextInt());

      insert(newNode);
```

```
}

private void insert(Node newNode) {
   if (root == null) {
      root = newNode;
   } else {
      insert(root, newNode);
   }
}

private void insert(Node current, Node newNode) {
   if (newNode.priority > current.priority) {
      if (current.left == null) {
         current.left = newNode;
      } else {
         insert(current.left, newNode);
      }
   } else {
      if (current.right == null) {
         current.right = newNode;
      } else {
         insert(current.right, newNode);
```

```
        }
    }

    rotate(current);
}

private void rotate(Node current) {
    if (current.left != null && current.right != null) {
        if (current.left.priority > current.right.priority) {
            Node leftChild = current.left;
            current.left = leftChild.right;
            leftChild.right = current;

            rotate(leftChild);
        }
    }
}

public void delete(int value) {
    if (root == null) {
        return;
    }
```

```
    delete(root, value);
}

private void delete(Node current, int value) {
    if (current.value == value) {
        if (current.left == null && current.right == null) {
            root = null;
        } else if (current.left == null) {
            root = current.right;
        } else if (current.right == null) {
            root = current.left;
        } else {
            delete(current.left, value);
        }
    } else if (current.value < value) {
        delete(current.right, value);
    } else {
        delete(current.left, value);
    }

    rotate(current);
```

```
}

public boolean contains(int value) {
    if (root == null) {
        return false;
    }

    return contains(root, value);
}

private boolean contains(Node current, int value) {
    if (current.value == value) {
        return true;
    } else if (current.value < value) {
        return contains(current.right, value);
    } else {
        return contains(current.left, value);
    }
}

private static class Node {
```

```
        private int value;
        private int priority;
        private Node left;
        private Node right;

        public Node(int value, int priority) {
            this.value = value;
            this.priority = priority;
        }
    }
}
```

This is just a simple example of how to implement a treap in Java. There are many other ways to implement it, and there are also many different ways to optimize it.

Link-Cut Trees

A link-cut tree is a data structure that can be used to represent a forest, a set of rooted trees. It supports the following operations:

- Link : Attach a tree consisting of a single node to the forest.
- Cut : Disconnect a node (and its subtree) from the tree of which it is part.
- Find-root : Given a node, find the root of the tree to which it belongs.

Link-cut trees are a powerful data structure that can be used to solve a variety of problems, such as dynamic connectivity, minimum spanning trees, and range minimum queries.

Here is an example of how to implement a link-cut tree in Java:

```
class LinkCutTree {


   private static final int MAX_LEVELS = 32;


   private Node[] nodes;


   public LinkCutTree() {
      nodes = new Node[MAX_LEVELS];
   }


   public void link(int u, int v) {
      nodes[u].parent = v;
```

```
    nodes[v].children.add(u);
}

public void cut(int u) {
    nodes[u].parent = -1;
    nodes[nodes[u].parent].children.remove(u);
}

public int findRoot(int u) {
    while (nodes[u].parent != -1) {
        u = nodes[u].parent;
    }

    return u;
}

private static class Node {

    int value;
    int parent;
    List<Integer> children;
```

```
        public Node(int value) {
            this.value = value;
            this.parent = -1;
            this.children = new ArrayList<>();
        }
    }
}
```

This is just a simple example of how to implement a link-cut tree in Java. There are many other ways to implement it, and there are also many different ways to optimize it.

Here are some of the advantages of link-cut trees:

- They are very efficient for dynamic connectivity problems.
- They can be used to solve a variety of other problems, such as minimum spanning trees and range minimum queries.
- They are relatively easy to implement.

Here are some of the disadvantages of link-cut trees:

- They can be more complex than other data structures, such as binary search trees.
- They can be more difficult to understand and debug.

Overall, link-cut trees are a powerful and versatile data structure that can be used to solve a variety of problems.