Data Structures and Algorithms

Assignment 1

Red-Black Tree:

Memory usage: **O(n)**

Time complexity:

|  |  |  |
| --- | --- | --- |
| Function | Best case | Worst case |
| Insert | **O**(**log**(**n**)) | **O**(**log**(**n**)) |
| Search | **O**(**log**(**n**)) | **O**(**log**(**n**)) |
| Delete | **O**(**log**(**n**)) | **O**(**log**(**n**)) |

Node structure:

Изображение выглядит как текст

Автоматически созданное описание

For balancing tree, we use such methods, like:

**1**. **Left Rotate**:

In left rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

**Initial tree**:

Изображение выглядит как текст, часы, датчик

Автоматически созданное описание

**Left Rotate:**

**Изображение выглядит как текст, часы, датчик

Автоматически созданное описание**

**2**. **Right Rotate**:

In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

**Initial tree**:

Изображение выглядит как текст, часы, датчик

Автоматически созданное описание

**Right Rotate:**

**Изображение выглядит как текст, часы, датчик

Автоматически созданное описание**

AVL Tree:

Memory usage: **O(n)**

Time complexity:

|  |  |  |
| --- | --- | --- |
| Function | Best case | Worst case |
| Insert | **O**(**log**(**n**)) | **O**(**log**(**n**)) |
| Search | **O**(**log**(**n**)) | **O**(**log**(**n**)) |
| Delete | **O**(**log**(**n**)) | **O**(**log**(**n**)) |

Node structure:

Изображение выглядит как текст

Автоматически созданное описание

For balancing tree, we use such methods, like:

**1**. **Left Rotate**:

In left rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

**Initial tree**:

Изображение выглядит как текст, часы, датчик

Автоматически созданное описание

**Left Rotate:**

**Изображение выглядит как текст, часы, датчик

Автоматически созданное описание**

**2**. **Right Rotate**:

In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

**Initial tree**:

Изображение выглядит как текст, часы, датчик

Автоматически созданное описание

**Right Rotate:**

**Изображение выглядит как текст, часы, датчик

Автоматически созданное описание**

HashMap:

Memory usage: **O(n)**

Time complexity:

|  |  |  |
| --- | --- | --- |
| Function | Best case | Worst case |
| Insert | **O**(**1**) | **O**(**n**) |
| Search | **O**(**1**) | **O**(**n**) |
| Delete | **O**(**1**) | **O**(**n**) |

A hash table is a data structure that associates keys with values. Hash tables provide constant time searches **O**(**1**) on average, regardless of the number of items in the table. But a rare worst case scenario can flat to **O**(**n**). Compared to other data structures, associative arrays are hash tables most useful for storing large amounts of data. Hash tables are much faster than self-balancing **BST**.

My hash function:

Изображение выглядит как текст

Автоматически созданное описание

**Collision resolutions**:

In my project I’ve used two types of collision resolutions, such as:

1. **Chaining method**
2. **Linear probing**

**Chaining method:**

Each cell of the array H is a pointer to a linked list (chain) of key-value pairs corresponding to the same key hash value. Collisions simply result in chains that are longer than one element.

Finding or deleting an element requires searching through all the elements of the corresponding chain to find an element with a given key in it. To add an element, you need to add an element to the end or the beginning of the corresponding list, and if the fill factor becomes too large, increase the size of the array **H** and rebuild the table.

Node structure:

Изображение выглядит как текст

Автоматически созданное описание

**Linear probing:**

The array H stores the key-value pairs themselves. The element insertion algorithm checks the cells of the array H in some order until the first free cell is found, in which the new element will be written. This ordering is computed on the fly, saving on memory for the pointers required in chained hash tables.

**Time testing:**

Every test have been done 100 times to find average time or memory usage.

Search test – I’ve been looking for number 31050405 which was at the end of BST or HashMap. The same I’ve done in Delete test.

**Red-Black Tree:**

|  |  |
| --- | --- |
| Dataset | Memory usage(**MB**) |
| 1.000.000 | 104 |
| 10.000.000 | 567 |
| 15.000.000 | 773 |
| 20.000.000 | 1031 |

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset | Insert | Search | Delete |
| 1.000.000 | 0,781 | 0,000025 | 0,00008 |
| 10.000.000 | 8,983 | 0,00003 | 0,000151 |
| 15.000.000 | 13,903 | 0,000023 | 0,000151 |
| 20.000.000 | 20,036 | 0,000029 | 0,000114 |

**AVL Tree:**

|  |  |
| --- | --- |
| Dataset | Memory usage(**MB**) |
| 1.000.000 | 106 |
| 10.000.000 | 514 |
| 15.000.000 | 770 |
| 20.000.000 | 1033 |

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset | Insert | Search | Delete |
| 1.000.000 | 0,949 | 0,000027 | 0,000114 |
| 10.000.000 | 9,823 | 0,000021 | 0,000178 |
| 15.000.000 | 16,828 | 0,000024 | 0,000074 |
| 20.000.000 | 21,987 | 0,000015 | 0,000128 |

**HashMap(Chaining):**

|  |  |
| --- | --- |
| Dataset | Memory usage(**MB**) |
| 1.000.000 | 124 |
| 10.000.000 | 1057 |
| 15.000.000 | 1693 |
| 20.000.000 | 2193 |

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset | Insert | Search | Delete |
| 1.000.000 | 0,243 | 0,000012 | 0,000062 |
| 10.000.000 | 4,029 | 0,000012 | 0,000064 |
| 15.000.000 | 7,924 | 0,000008 | 0,000072 |
| 20.000.000 | 9,206 | 0,000004 | 0,000068 |

**HashMap(Linear):**

|  |  |
| --- | --- |
| Dataset | Memory usage(**MB**) |
| 1.000.000 | 115 |
| 10.000.000 | 986 |
| 15.000.000 | 1640 |
| 20.000.000 | 2109 |

|  |  |  |  |
| --- | --- | --- | --- |
| Dataset | Insert | Search | Delete |
| 1.000.000 | 0,196 | 0,107657 | 0,007386 |
| 10.000.000 | 4,397 | 0,000004 | 0,000009 |
| 15.000.000 | 8,69 | 0,000005 | 0,000009 |
| 20.000.000 | 10,668 | 0,000061 | 0,000005 |

**Comparison:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | RB Tree | AVL Tree | HashMap(Chain) | HashMap(Linear) |
| 1.000.000 | **104** | 106 | **124** | 115 |
| 10.000.000 | 567 | **514** | **1057** | 986 |
| 15.000.000 | 773 | **770** | **1693** | 1640 |
| 20.000.000 | **1031** | 1033 | **2193** | 2109 |

Here we can see, that BST’s use less memory than HashMap’s.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | RB Tree | AVL Tree | HashMap(Chain) | HashMap(Linear) |
| 1.000.000 | 0,781 | **0,949** | 0,243 | **0,196** |
| 10.000.000 | 8,983 | **9,823** | **4,029** | 4,397 |
| 15.000.000 | 13,903 | 16,828 | **7,924** | 8,69 |
| 20.000.000 | 20,036 | 21,987 | **9,206** | 10,668 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | RB Tree | AVL Tree | HashMap(Chain) | HashMap(Linear) |
| 1.000.000 | 0,000025 | 0,000027 | **0,000012** | **0,107657** |
| 10.000.000 | **0,00003** | 0,000021 | 0,000012 | **0,000004** |
| 15.000.000 | 0,000023 | **0,000024** | 0,000008 | **0,000005** |
| 20.000.000 | 0,000029 | 0,000015 | **0,000004** | **0,000061** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dataset | **RB Tree** | **AVL Tree** | **HashMap(Chain)** | **HashMap(Linear)** |
| 1.000.000 | 0,00008 | 0,000114 | **0,000062** | **0,007386** |
| 10.000.000 | 0,000151 | **0,000178** | 0,000064 | **0,000009** |
| 15.000.000 | **0,000151** | 0,000074 | 0,000072 | **0,000009** |
| 20.000.000 | 0,000114 | **0,000128** | 0,000068 | **0,000005** |

**Conclusion:**

A field of tests made sure that **Hash tables** are faster than **BSTs**, but **they take up much more memory**. **BSTs** are much **more stable**, they always stick to **O**(**log**(**n**)), table hashes can change from **O**(**1**) to **O**(**n**) in turn. I also realized that **the speed of code execution** also **depends on the input data**, this can be seen when comparing a data set for 1 million data and others. A dataset with 1 million values ​​is much slower than other datasets.

If you need speed and the amount of memory consumed is not important for you, then your choice is the **Hash table**. If the amount of memory consumed is important to you - **BST**.