Analysis document

Word Searching

Nhat Nam Ha

Contents

Introduction

Word searching is a task counting the number of occurrences of a given word in sentence or document. This assignment required to measure time performance implementation by using linear searching and using binary search tree data structure. Later, another data structure can be measured is hash table.

Implementation

Since the assignment does not require implementing everything from beginning, example binary search tree and hash table by hand. There is implementation for these data structures in standard template library (STL) available in most programming language, avoid "reinventing the wheel" and it is best practices to use library in most projects as the correctness has been tested.

My main choice is C#, there are 2 collections are:

- SortedDictionary: Binary Search Tree based collection.
- Dictionary: Hash Table based collection.

8

1. If you require sorted set, use `SortedDictionary<T,U>`. This is implemented using a binary search tree. Admittedly, you will be using 64-bits per entry because you are storing a key-value pair underneath. You can write a wrapper around it like this:

```
class Set<T> : SortedDictionary<T, bool>
{
    public void Add(T item)
    {
        this.Add(item, true);
    }
}
```

2. If you don't require a sorted set, use `HashSet<T>`.
3. Otherwise, check out C5 Generic Collection Library. In particular `TreeSet<T>`. It is a red-black tree and only stores the values.

Share Follow

answered Feb 23, 2009 at 22:18

Szymon Rozga

18k ● 7 ● 54 ● 66

Figure 1. https://stackoverflow.com/questions/575406/what-is-the-c-sharp-equivalent-of-the-stl-set

These are the implementation for word searching using the C# STL library.

```
15 references
public class WordSearch
{
    private string text;
    private char[] delimiterChars = { ' ', ',', '.', ':', '\t' };

    List<string> words;
    SortedDictionary<string, long> binaryTree;
    Dictionary<string, long> hashTable;

    7 references | 6/6 passing
    public WordSearch(string text)
    {
        this.text = text;
        binaryTree = new();
        hashTable = new();

        words = this.text.Split(delimiterChars).ToList();

        foreach (string sample in words)
        {
            if (!binaryTree.ContainsKey(sample)) binaryTree[sample] = 0;
            if (!hashTable.ContainsKey(sample)) hashTable[sample] = 0;
            binaryTree[sample]++;
            hashTable[sample]++;
        }
    }

    4 references | 2/2 passing
    public long LinearSearch(string word)
    {
        long result = 0;
        foreach (string sample in words)
        {
            if (sample.Equals(word)) result++;
        }
        return result;
    }

    5 references | 4/4 passing
    public long BinarySearch(string word)
    {
        bool existed = binaryTree.TryGetValue(word, out long result);
        return existed ? result : 0;
    }

    1 reference
    public long HashSearch(string word)
    {
        bool existed = hashTable.TryGetValue(word, out long result);
        return existed ? result : 0;
    }
}
```

Unit tests

For each implementation, I tested for 4 scenarios:

- There is a word existed in a sentence or document, return number of occurrences.
- Return 0 if not existed
- Check both upper-case and lower-case word search

```
[Test]
| 0 references
public void LinearSuccess()
{
    WordSearch item = new("who am I and some random thing from me, who ?");
    var result = item.LinearSearch("who");
    Assert.AreEqual(result, 2);
}

[Test]
| 0 references
public void LinearCannotFind()
{
    WordSearch item = new("who am I and some random thing from me, who ?");
    var result = item.LinearSearch("whommmmm");
    Assert.AreEqual(result, 0);
}

[Test]
| 0 references
public void LinearCaseSensitive1()
{
    WordSearch item = new("Who am I, who who who ?");
    var result = item.LinearSearch("Who");
    Assert.AreEqual(result, 1);
}

[Test]
| 0 references
public void LinearCaseSensitive2()
{
    WordSearch item = new("Who am I, who who who ?");
    var result = item.LinearSearch("who");
    Assert.AreEqual(result, 3);
}
```

```
[Test]
0 references
public void BinarySuccess()
{
    WordSearch item = new("who am I and some random thing from me, who ?");
    var result = item.BinarySearch("who");
    Assert.AreEqual(result, 2);
}

[Test]
0 references
public void BinaryCannotFind()
{
    WordSearch item = new("who am I and some random thing from me, who ?");
    var result = item.BinarySearch("whommmmm");
    Assert.AreEqual(result, 0);
}

[Test]
0 references
public void BinaryCaseSensitive1()
{
    WordSearch item = new("Who am I, who who who ?");
    var result = item.BinarySearch("Who");
    Assert.AreEqual(result, 1);
}

[Test]
0 references
public void BinaryCaseSensitive2()
{
    WordSearch item = new("Who am I, who who who ?");
    var result = item.BinarySearch("who");
    Assert.AreEqual(result, 3);
}
```

```
[Test]
0 references
public void HashSuccess()
{
    WordSearch item = new("who am I and some random thing from me, who ?");
    var result = item.HashSearch("who");
    Assert.AreEqual(result, 2);
}

[Test]
0 references
public void HashCannotFind()
{
    WordSearch item = new("who am I and some random thing from me, who ?");
    var result = item.HashSearch("whommmmm");
    Assert.AreEqual(result, 0);
}

[Test]
0 references
public void HashCaseSensitive1()
{
    WordSearch item = new("Who am I, who who who ?");
    var result = item.HashSearch("Who");
    Assert.AreEqual(result, 1);
}

[Test]
0 references
public void HashCaseSensitive2()
{
    WordSearch item = new("Who am I, who who who ?");
    var result = item.HashSearch("who");
    Assert.AreEqual(result, 3);
}
```

BinaryCannotFind	20 ms	HashCaseSensitive2	< 1 ms
BinaryCaseSensitive1	< 1 ms	HashSuccess	< 1 ms
BinaryCaseSensitive2	< 1 ms	LinearCannotFind	< 1 ms
BinarySuccess	< 1 ms	LinearCaseSensitive1	< 1 ms
HashCannotFind	< 1 ms	LinearCaseSensitive2	< 1 ms
HashCaseSensitive1	< 1 ms	LinearSuccess	< 1 ms

Performance tests

Complexity analysis

The complexity of word searching mostly depends on the time complexity of data structures used.

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(1)	O(1)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(log N)	O(log N)	O(log N)	O(log N)

Figure 2 https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/

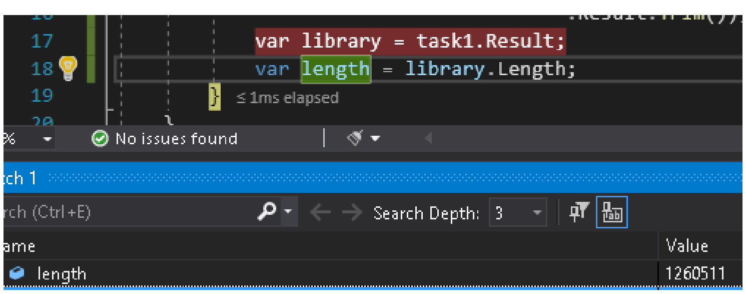
From the reference above, we extracted the complexity of insert and search of three data-structures are array, binary search tree and hash table as the scope of the assignment.

Operations/ Strategies	Linear Search	Binary Search Tree	Hash Table
Insert	O(1)	O(log N)	O(1)
Search	O(N)	O(log N)	O(1)

We expected that the time measurement in descending is linear search, binary search tree and hash table.

Benchmark

My benchmark measure time performance by number of queries to each implementation, loading a book and a list of the words for searching. The book has 1_260_511 characters inside.



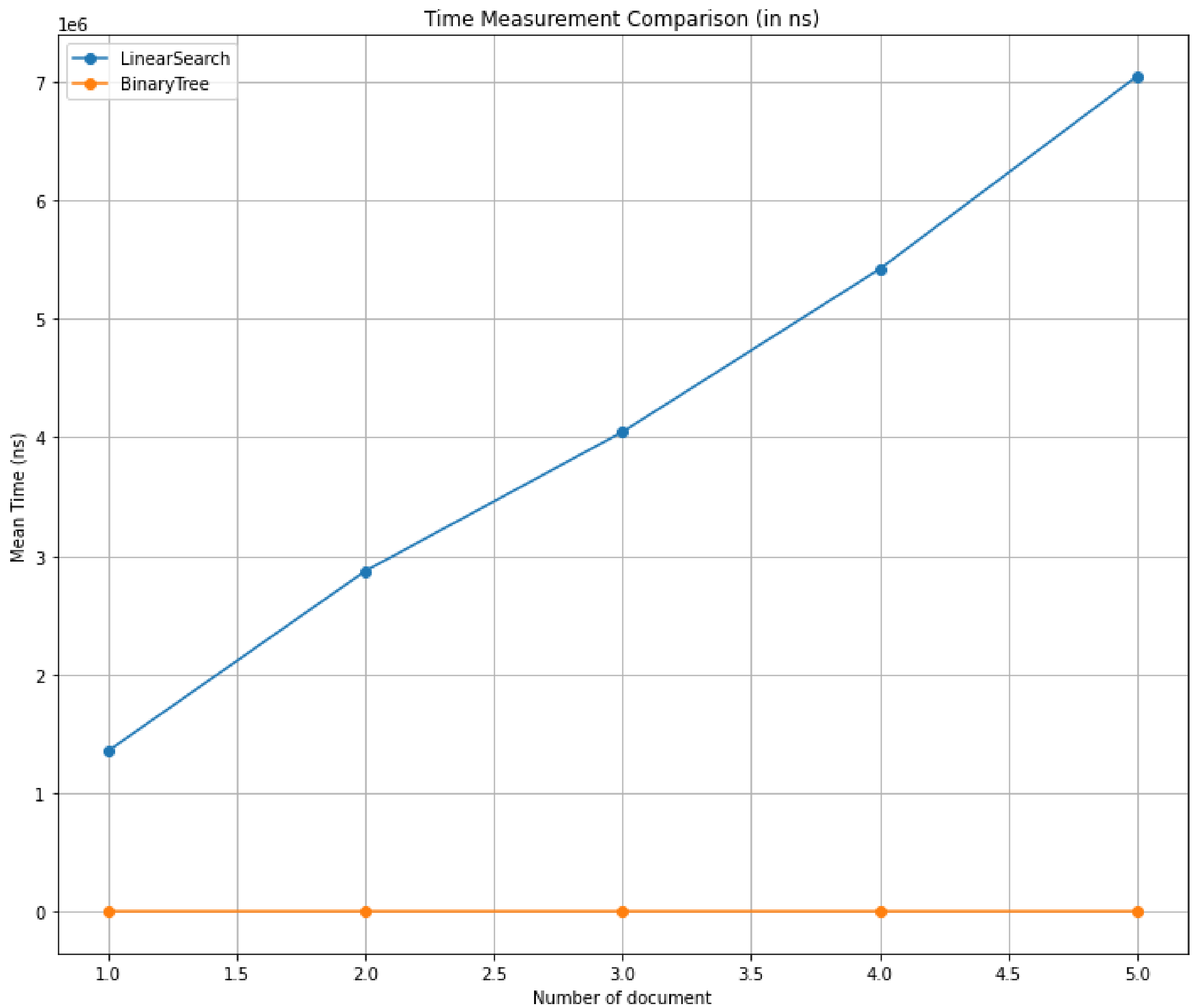
```
[Params(1, 2, 3, 4, 5)]
public int nrDocument;

private string library;
private string[] words;
private WordSearch[] wordSearchs;

[GlobalSetup]
0 references
public void Setup()
{
    wordSearchs = new WordSearch[5];
    var task1 = Task.Run(() => File.ReadAllTextAsync("Text2.txt")
                                .Result.Trim());
    var task2 = File.ReadAllLinesAsync("Search.txt");
    library = task1.Result;
    words = task2.Result;

    StringBuilder[] sb = new StringBuilder[5];

    for (int i = 0; i < 5; i++)
    {
        sb[i] = new(library);
        sb[i].AppendLine();
        if (i > 0) sb[i].Append(sb[i - 1]);
    }

    for (int i = 0; i < 5; i++)
    {
        var lib = sb[i].ToString();
        wordSearchs[i] = new(lib);
    }
}
```

```
[Benchmark(Baseline = true)]
0 references
public void LinearSearch()
{
    var _ = wordSearchs[nrDocument-1]
            .LinearSearch(words[0]);
}

[Benchmark]
0 references
public void BinaryTree()
{
    var _ = wordSearchs[nrDocument - 1]
            .BinarySearch(words[0]);
}

[Benchmark]
0 references
public void HashTable()
{
    var _ = wordSearchs[nrDocument - 1]
            .HashSearch(words[0]);
}
```

For variety length of document, my strategies is to make document longer by appending the whole previous documents, first WordSearch instance will search on one book, second WordSearch instance will search on two books, third WordSearch instance will search on three books, …

Method	nrDocument	Mean	Error	StdDev	Ratio	Allocated	Alloc Ratio
LinearSearch	1	1,368,814.1 ns	9,017.41 ns	8,434.89 ns	1.000	-	NA
BinaryTree	1	534.0 ns	2.61 ns	2.44 ns	0.000	-	NA
LinearSearch	2	2,398,083.7 ns	19,245.08 ns	18,001.86 ns	1.000	3 B	1.00
BinaryTree	2	554.1 ns	2.60 ns	2.30 ns	0.000	-	0.00
LinearSearch	3	4,067,758.4 ns	36,799.67 ns	34,422.43 ns	1.000	5 B	1.00
BinaryTree	3	580.4 ns	5.65 ns	5.29 ns	0.000	-	0.00
LinearSearch	4	5,175,514.9 ns	53,924.08 ns	47,802.30 ns	1.000	7 B	1.00
BinaryTree	4	524.6 ns	5.50 ns	5.14 ns	0.000	-	0.00
LinearSearch	5	6,521,922.2 ns	80,367.89 ns	75,176.17 ns	1.000	5 B	1.00
BinaryTree	5	562.3 ns	4.59 ns	4.29 ns	0.000	-	0.00

From the table above, Binary Search Tree performance faster more than 1000 times compares to Linear Search approach.



From the chart above, since the complexity of Linear Search is linear, it is noticing the line chart of linear search is linear and go up when the length of document increase.

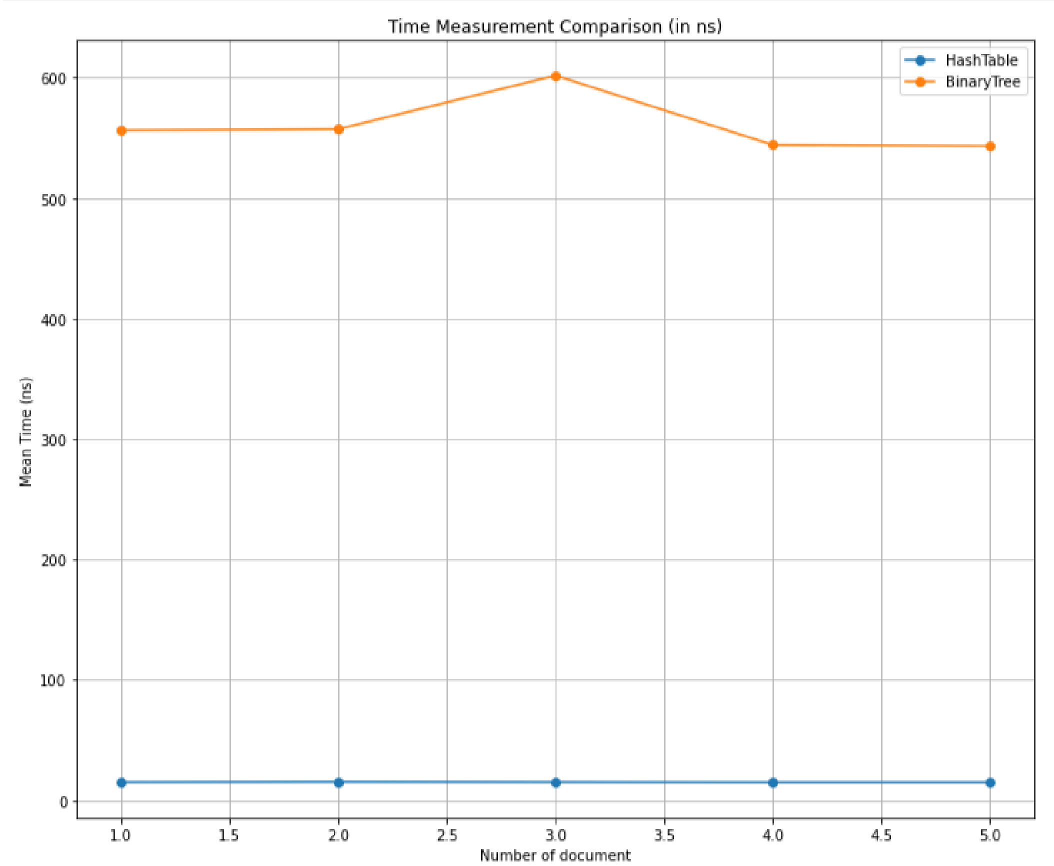
Consider using hash table for word searching, this is the benchmark code:

```
[Benchmark]
0 references
public void HashTable()
{
    for (int i = 0; i < nrWords; i++)
    {
        var _ = wordSearch.HashSearch(words[i]);
    }
}
```

For hash table comparison, it is better to compare with binary search tree for graph visualization.

Method	nrDocument	Mean	Error	StdDev	Ratio	Allocated	Alloc Ratio
LinearSearch	1	1,356,076.94 ns	2,484.007 ns	2,323.541 ns	1.000	1 B	1.00
BinaryTree	1	556.18 ns	2.422 ns	2.147 ns	0.000	-	0.00
HashTable	1	14.88 ns	0.151 ns	0.141 ns	0.000	-	0.00
LinearSearch	2	2,872,472.21 ns	12,370.636 ns	10,330.041 ns	1.000	3 B	1.00
BinaryTree	2	557.11 ns	4.379 ns	4.096 ns	0.000	-	0.00
HashTable	2	15.17 ns	0.135 ns	0.126 ns	0.000	-	0.00
LinearSearch	3	4,045,357.92 ns	26,208.649 ns	23,233.284 ns	1.000	13 B	1.00
BinaryTree	3	601.72 ns	4.396 ns	4.112 ns	0.000	-	0.00
HashTable	3	14.94 ns	0.119 ns	0.106 ns	0.000	-	0.00
LinearSearch	4	5,418,243.70 ns	37,053.075 ns	34,659.469 ns	1.000	7 B	1.00
BinaryTree	4	544.07 ns	3.875 ns	3.625 ns	0.000	-	0.00
HashTable	4	14.74 ns	0.144 ns	0.134 ns	0.000	-	0.00
LinearSearch	5	7,041,625.26 ns	23,666.360 ns	22,137.528 ns	1.000	5 B	1.00
BinaryTree	5	543.18 ns	3.679 ns	3.441 ns	0.000	-	0.00
HashTable	5	14.74 ns	0.098 ns	0.092 ns	0.000	-	0.00

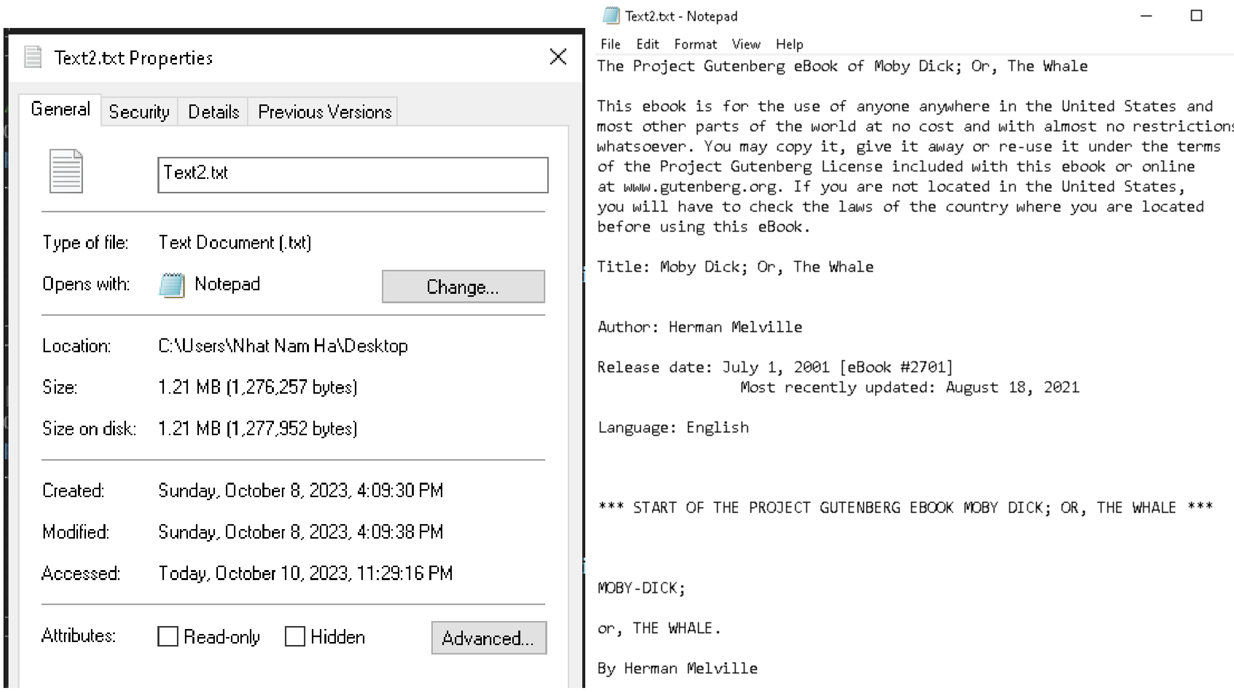
Hash table perform better than binary search tree almost 40 times based on the summarize table.



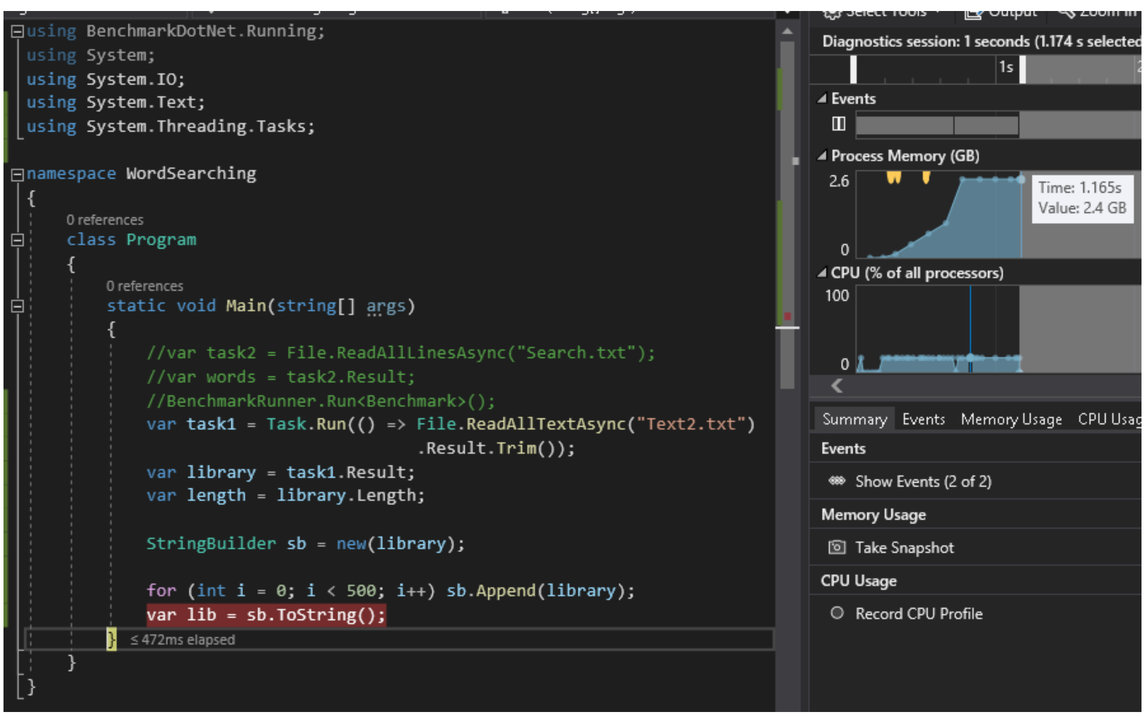
Potential issues

Size of the document

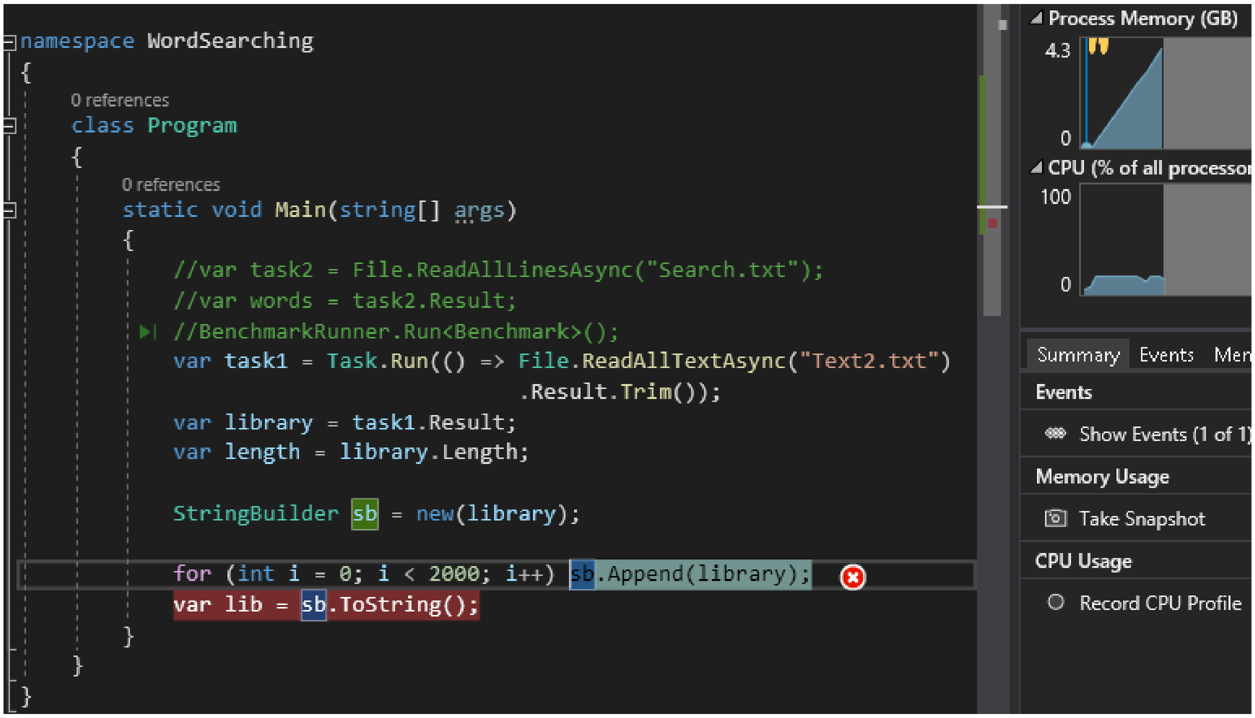
From the performance tests, the text file was used for the tests getting from the Gothenburg collections and it is in simple text format.



Test the limitation of my implementation by appending this document and check for memory management when debugging.



After appending 500 times of same book, our program hit at 2.4 GB of Ram memory, since my current laptop has 16 GB of Ram, so in theory, if appending 3000 times then program crashes.



After appending 2000 times, program crashes when using 4.3 GB of memory, it crashes because of during the benchmark, my computer did not have enough memory for allocation.

Diversity of words in document

This is an interesting potential issue, it is observed that when searching, we want to search the meaningful English words, the best way to find the collection of words is through the dictionary, in English will be Oxford dictionary.

The English Dictionary

First, let's look at how many words are in the Dictionary. The Second Edition of the 20-volume Oxford English Dictionary contains full entries for 171,476 words in current use (and 47,156 obsolete words). Webster's Third New International Dictionary, Unabridged, together with its 1993 Addenda Section, includes some 470,000 entries. But, the number of words in the Oxford and Webster Dictionaries are not the same as the number of words in English.

Figure 3 https://wordcounter.io/blog/how-many-words-are-in-the-english-language

Based on the information above, there are currently more than 200_000 different English words in Oxford dictionary. Luckily, based on our implementation, it is possible to have many entries above without memory allocation issue.

Suggestion improvements

We notice the program crashes when loading big chunk of string into memory, consider the diversity of words, first suggested solution is reading word by word in a file instead of loading the whole document to the memory solve the memory issue and keep the correctness of the program.

Here's my implementation of lazy extension to `StreamReader`. The idea is not to load the entire file into memory especially if your file is a single long line.

```
public static string ReadWord(this StreamReader stream, Encoding encoding)
{
    string word = "";
    // read single character at a time building a word
    // until reaching whitespace or (-1)
    while(stream.Read()
       .With(c => { // with each character . . .
            // convert read bytes to char
            var chr = encoding.GetChars(BitConverter.GetBytes(c)).First();

            if (c == -1 || Char.IsWhiteSpace(chr))
                 return -1; //signal end of word
            else
                 word = word + chr; //append the char to our word

            return c;
    }) > -1);  // end while(stream.Read() if char returned is -1
    return word;
}

public static T With<T>(this T obj, Func<T,T> f)
{
    return f(obj);
}
```

to use simply:

```
using (var s = File.OpenText(file))
{
    while(!s.EndOfStream)
        s.ReadWord(Encoding.Default).ToCharArray().DoSomething();
}
```

Share Improve this answer Follow edited May 12, 2014 at 15:41 answered Feb 15, 2014 at 1:06

K. R.

1,240 ●17 ●20

Figure 4 https://stackoverflow.com/questions/15229780/how-to-read-the-text-word-by-word

We also have another approach for text searching beside binary search tree and hash table is the Trie, with the observed that there are many words are prefix of the longer one, for example:

- "Sam" is a prefix of "Same"
- "And" is a prefix of "Andy"
- "Mount" is a prefix of "Mountain"

In computer science, a **trie** (/ˈtriː/, /ˈtraɪ/), also called **digital tree** or **prefix tree**,[1] is a type of *k*-ary search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In order to access a key (to recover its value, change it, or remove it), the trie is traversed depth-first, following the links between nodes, which represent each character in the key.

Figure 5 https://en.wikipedia.org/wiki/Trie

For both insert and search operation, the time complexity depends on the length of the word (number of characters in word) we want to find, most common words have the length around 5 to 10 characters.