Symbol Tables

Outline

1. What is a Symbol Table?

2. API

3. Implementations

4. Performance Characteristics

What is a Symbol Table?

What is a Symbol Table?

A symbol table is a data structure for key-value pairs that supports two operations: insert (put) a new pair into the table and search (get) the value associated with a given key

What is a Symbol Table?

A symbol table is a data structure for key-value pairs that supports two operations: insert (put) a new pair into the table and search (get) the value associated with a given key

Applications

Application	Purpose	Key	Value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
file share	find song to download	name of song	computer ID
web search	find relevant web pages	keyword	list of page names
compiler	find type and value	variable name	type and value

What is a Symbol Table?

What is a Symbol Table?

Conventions:

What is a Symbol Table?

Conventions:

- No duplicate keys are allowed; when a client puts a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one

What is a Symbol Table?

Conventions:

- No duplicate keys are allowed; when a client puts a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one
- Keys/values must not be `null`

What is a Symbol Table?

Conventions:

- No duplicate keys are allowed; when a client puts a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one
- Keys/values must not be `null`
- Deleting a key involves removing the key (and the associated value) from the table immediately

API

API

BasicST<Key, Value>

`boolean isEmpty()`	returns `true` if this symbol table is empty, and `false` otherwise
`int size()`	returns the number of key-value pairs in this symbol table
`void put(Key key, Value value)`	inserts the `key` and `value` pair into this symbol table
`Value get(Key key)`	returns the value associated with `key` in this symbol table, or `null`
`boolean contains(Key key)`	returns `true` if this symbol table contains `key`, and `false` otherwise
`void delete(Key key)`	deletes `key` and the associated value from this symbol table
`Iterable<Key> keys()`	returns all the keys in this symbol table

API

API

☰ *OrderedST<Key extends Comparable<Key>, Value>*

`boolean isEmpty()`	returns `true` if this symbol table is empty, and `false` otherwise
`int size()`	returns the number of key-value pairs in this symbol table
`void put(Key key, Value value)`	inserts the `key` and `value` pair into this symbol table
`Value get(Key key)`	returns the value associated with `key` in this symbol table, or `null`
`boolean contains(Key key)`	returns `true` if this symbol table contains `key`, and `false` otherwise
`void delete(Key key)`	deletes `key` and the associated value from this symbol table
`Iterable<Key> keys()`	returns all the keys in this symbol table in sorted order
`Key min()`	returns the smallest key in this symbol table
`Key max()`	returns the largest key in this symbol table
`void deleteMin()`	deletes the smallest key and the associated value from this symbol table
`void deleteMax()`	deletes the largest key and the associated value from this symbol table
`Key floor(Key key)`	returns the largest key in this symbol table that is smaller than or equal to `key`
`Key ceiling(Key key)`	returns the smallest key in this symbol table that is greater than or equal to `key`
`int rank(Key key)`	returns the number of keys in this symbol table that are strictly smaller than `key`
`Key select(int k)`	returns the key in this symbol table with the rank `k`
`int size(Key lo, Key hi)`	returns the number of keys in this symbol table that are in the interval `[lo, hi]`
`Iterable<Key> keys(Key lo, Key hi)`	returns the keys in this symbol table that are in the interval `[lo, hi]` in sorted order

API

API

Program: `FrequencyCounter.java`

API

Program: `FrequencyCounter.java`

- Command-line input: *minLen* (int)

API

Program: `FrequencyCounter.java`

- Command-line input: *minLen* (int)
- Standard input: sequence of words

API

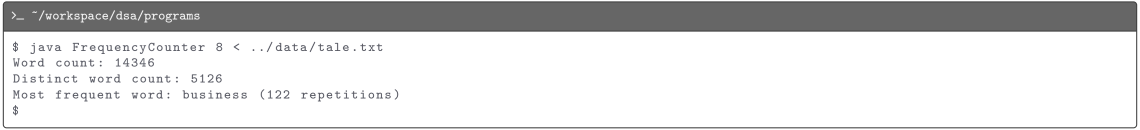
Program: `FrequencyCounter.java`

- Command-line input: *minLen* (int)
- Standard input: sequence of words
- Standard output: for the words that are at least as long as *minLen*, the total word count, the number of distinct words, and the most frequent word

API

Program: `FrequencyCounter.java`

- Command-line input: *minLen* (int)
- Standard input: sequence of words
- Standard output: for the words that are at least as long as *minLen*, the total word count, the number of distinct words, and the most frequent word



```
>_ ~/workspace/dsa/programs
```

```
$ java FrequencyCounter 8 < ../data/tale.txt
Word count: 14346
Distinct word count: 5126
Most frequent word: business (122 repetitions)
$
```

API

API

FrequencyCounter.java

```java
import dsa.SeparateChainingHashST;
import stdlib.StdIn;
import stdlib.StdOut;

public class FrequencyCounter {
    public static void main(String[] args) {
        SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<>();
        int minLen = Integer.parseInt(args[0]);
        int distinct = 0, words = 0;
        while (!StdIn.isEmpty()) {
            String key = StdIn.readString();
            if (key.length() < minLen) {
                continue;
            }
            words++;
            if (st.contains(key)) {
                st.put(key, st.get(key) + 1);
            } else {
                st.put(key, 1);
                distinct++;
            }
        }
        int maxFreq = 0;
        String maxFreqWord = "";
        for (String word : st.keys()) {
            if (st.get(word) > maxFreq) {
                maxFreq = st.get(word);
                maxFreqWord = word;
            }
        }
        StdOut.println("Word count: " + words);
        StdOut.println("Distinct word count: " + distinct);
        StdOut.printf("Most frequent word: %s (%d repetitions)\n", maxFreqWord, maxFreq);
    }
}
```

Implementations · LinearSearchST

Implementations · LinearSearchST

LinearSearchST.java

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

public class LinearSearchST<Key, Value> implements BasicST<Key, Value> {
    private Node first;
    private int n;

    public LinearSearchST() {
        first = null;
        n = 0;
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return n;
    }

    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
            throw new IllegalArgumentException("value is null");
        }
        for (Node x = first; x != null; x = x.next) {
            if (key.equals(x.key)) {
                x.value = value;
                return;
            }
        }
```

Implementations · LinearSearchST

LinearSearchST.java

```java
        first = new Node(key, value, first);
        n++;
    }

    public Value get(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        for (Node x = first; x != null; x = x.next) {
            if (key.equals(x.key)) {
                return x.value;
            }
        }
        return null;
    }

    public boolean contains(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        return get(key) != null;
    }

    public void delete(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        first = delete(first, key);
    }

    public Iterable<Key> keys() {
        LinkedQueue<Key> queue = new LinkedQueue<Key>();
        for (Node x = first; x != null; x = x.next) {
            queue.enqueue(x.key);
        }
```

Implementations · LinearSearchST

LinearSearchST.java

```java
        return queue;
    }

    private class Node {
        private Key key;
        private Value value;
        private Node next;

        public Node(Key key, Value value, Node next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    private Node delete(Node x, Key key) {
        if (x == null) {
            return null;
        }
        if (key.equals(x.key)) {
            n--;
            return x.next;
        }
        x.next = delete(x.next, key);
        return x;
    }

    public static void main(String[] args) {
        LinearSearchST<String, Integer> st = new LinearSearchST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys()) {
            StdOut.println(s + " " + st.get(s));
```

Implementations · LinearSearchST

LinearSearchST.java

```
        }
    }
}
```

Implementations · BinarySearchST

Implementations · BinarySearchST

BinarySearchST.java

```java
package dsa;

import stdlib.StdIn;
import stdlib.StdOut;

import java.util.NoSuchElementException;

public class BinarySearchST<Key extends Comparable<Key>, Value>
        implements OrderedST<Key, Value> {
    private Key[] keys;
    private Value[] vals;
    private int n = 0;

    public BinarySearchST() {
        keys = (Key[]) new Comparable[2];
        vals = (Value[]) new Object[2];
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public int size() {
        return n;
    }

    public void put(Key key, Value value) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        if (value == null) {
            throw new IllegalArgumentException("value is null");
        }
        int i = rank(key);
        if (i < n && keys[i].compareTo(key) == 0) {
```

Implementations · BinarySearchST

BinarySearchST.java

```java
            vals[i] = value;
            return;
        }
        if (n == keys.length) {
            resize(2 * keys.length);
        }
        for (int j = n; j > i; j--) {
            keys[j] = keys[j - 1];
            vals[j] = vals[j - 1];
        }
        keys[i] = key;
        vals[i] = value;
        n++;
    }

    public Value get(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        int i = rank(key);
        if (i < n && keys[i].compareTo(key) == 0) {
            return vals[i];
        }
        return null;
    }

    public boolean contains(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        return get(key) != null;
    }

    public void delete(Key key) {
        if (key == null) {
```

Implementations · BinarySearchST

BinarySearchST.java

```java
            throw new IllegalArgumentException("key is null");
        }
        int i = rank(key);
        if (i == n || keys[i].compareTo(key) != 0) {
            return;
        }
        for (int j = i; j < n - 1; j++) {
            keys[j] = keys[j + 1];
            vals[j] = vals[j + 1];
        }
        n--;
        keys[n] = null;
        vals[n] = null;
        if (n > 0 && n == keys.length / 4) {
            resize(keys.length / 2);
        }
    }

    public Iterable<Key> keys() {
        return keys(min(), max());
    }

    public Key min() {
        if (isEmpty()) {
            throw new NoSuchElementException("Symbol table is empty");
        }
        return keys[0];
    }

    public Key max() {
        if (isEmpty()) {
            throw new NoSuchElementException("Symbol table is empty");
        }
        return keys[n - 1];
    }
```

Implementations · BinarySearchST

BinarySearchST.java

```java
    public void deleteMin() {
        if (isEmpty()) {
            throw new NoSuchElementException("Symbol table is empty");
        }
        delete(min());
    }

    public void deleteMax() {
        if (isEmpty()) {
            throw new NoSuchElementException("Symbol table is empty");
        }
        delete(max());
    }

    public Key floor(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        int i = rank(key);
        if (i < n && key.compareTo(keys[i]) == 0) {
            return keys[i];
        }
        if (i == 0) {
            return null;
        }
        return keys[i - 1];
    }

    public Key ceiling(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        int i = rank(key);
        if (i == n) {
```

Implementations · BinarySearchST

BinarySearchST.java

```java
            return null;
        }
        return keys[i];
    }

    public int rank(Key key) {
        if (key == null) {
            throw new IllegalArgumentException("key is null");
        }
        int lo = 0, hi = n - 1;
        while (lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int cmp = key.compareTo(keys[mid]);
            if (cmp < 0) {
                hi = mid - 1;
            } else if (cmp > 0) {
                lo = mid + 1;
            } else {
                return mid;
            }
        }
        return lo;
    }

    public Key select(int k) {
        if (k < 0 || k >= size()) {
            throw new IllegalArgumentException("Invalid rank");
        }
        return keys[k];
    }

    public int size(Key lo, Key hi) {
        if (lo == null) {
            throw new IllegalArgumentException("lo is null");
        }
```

Implementations · BinarySearchST

BinarySearchST.java

```java
        if (hi == null) {
            throw new IllegalArgumentException("hi is null");
        }
        if (lo.compareTo(hi) > 0) {
            return 0;
        }
        if (contains(hi)) {
            return rank(hi) - rank(lo) + 1;
        }
        return rank(hi) - rank(lo);
    }

    public Iterable<Key> keys(Key lo, Key hi) {
        if (lo == null) {
            throw new IllegalArgumentException("lo is null");
        }
        if (hi == null) {
            throw new IllegalArgumentException("hi is null");
        }
        LinkedQueue<Key> queue = new LinkedQueue<Key>();
        if (lo.compareTo(hi) > 0) {
            return queue;
        }
        for (int i = rank(lo); i < rank(hi); i++) {
            queue.enqueue(keys[i]);
        }
        if (contains(hi)) {
            queue.enqueue(keys[rank(hi)]);
        }
        return queue;
    }

    private void resize(int capacity) {
        Key[] tempKeys = (Key[]) new Comparable[capacity];
        Value[] tempVals = (Value[]) new Object[capacity];
```

Implementations · BinarySearchST

BinarySearchST.java

```java
        for (int i = 0; i < n; i++) {
            tempKeys[i] = keys[i];
            tempVals[i] = vals[i];
        }
        keys = tempKeys;
        vals = tempVals;
    }

    public static void main(String[] args) {
        BinarySearchST<String, Integer> st = new BinarySearchST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }
        for (String s : st.keys()) {
            StdOut.println(s + " " + st.get(s));
        }
    }
}
```

Performance Characteristics

Performance Characteristics

Operation	Unordered Linked List	Ordered Array
search	n	$\lg n$
insert	n	n
efficiently supports ordered operations?	no	yes