

Algorithms and Data Structures

Dictionaries

Marcin Sydow

Web Mining Lab
PJWSTK

Topics covered by this lecture:

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

- Dictionary
- Hashtable
- Binary Search Tree (BST)
- AVL Tree
- Self-organising BST

Dictionary

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

Dictionary is an abstract data structure that supports the following operations:

- `search(K key)`
(returns the value associated with the given key)¹
- `insert(K key, V value)`
- `delete(K key)`

Each element stored in a dictionary is identified by a key of type K . Dictionary represents a mapping from keys to values.

Dictionaries have numerous applications

¹Search can return a special value if key is absent in dictionary

Examples

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

- contact book
key: name of person; value: telephone number
- table of program variable identifiers
key: identifier; value: address in memory
- property-value collection
key: property name; value: associated value
- natural language dictionary
key: word in language X; value: word in language Y
- etc.

Implementations

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

- simple implementations: sorted or unsorted sequences, direct addressing
- hash tables
- binary search trees (BST)
- AVL trees
- self-organising BST
- red-black trees
- (a,b)-trees (in particular: 2-3-trees)
- B-trees
- and other ...

Simple implementations of Dictionary

Elements of a dictionary can be kept in a sequence (linked list or array):

(data size: number of elements (n); dom. op.: key comparison)

- unordered:
search: $O(n)$; insert: $O(1)$; delete: $O(n)$
- ordered array:
search: $O(\log n)$; insert $O(n)$; delete $O(n)$
- ordered linked list:
search: $O(n)$; insert $O(n)$; delete: $O(n)$

(keeping the sequence sorted does not help in this case!)

Space complexity: $\Theta(n)$

Direct Addressing

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

Assume potential keys are numbers from some universe $U \subseteq N$.

An element with key $k \in U$ can be kept under index k in a $|U|$ -element array:

search: $O(1)$; insert: $O(1)$; delete: $O(1)$

This is extremely fast! What is the price?

Direct Addressing

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

Assume potential keys are numbers from some universe $U \subseteq N$.

An element with key $k \in U$ can be kept under index k in a $|U|$ -element array:

search: $O(1)$; insert: $O(1)$; delete: $O(1)$

This is extremely fast! What is the price?

n - number of elements currently kept. What is space complexity?

Direct Addressing

Assume potential keys are numbers from some universe $U \subseteq N$.

An element with key $k \in U$ can be kept under index k in a $|U|$ -element array:

search: $O(1)$; insert: $O(1)$; delete: $O(1)$

This is extremely fast! What is the price?

n - number of elements currently kept. What is space complexity?

space complexity: $O(|U|)$ ($|U|$ can be very high, even if we keep a small number of elements!)

Direct addressing is fast but waists a lot of memory (when $|U| \gg n$)

Hashtables

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

The idea is simple.

Elements are kept in an m -element array $[0, \dots, m - 1]$, where $m \ll |U|$

The index of key is computed by fast **hash function**:

hashing function: $h : U \rightarrow [0..m - 1]$

For a given key k its position is computed by $h(k)$ before each dictionary operation.

Hashing Non-integer Keys

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

What if the type of key is not integer?

Additional step is needed: before computing the hash function, the key should be transformed to integer.

For example: key is a string of characters, the transformation should depend on all characters.

This transforming function should have similar properties to hashing function.

Hash Function

Important properties of an *ideal* hash function

$h \rightarrow [0, \dots, m - 1]$:

- **uniform load** on each index $0 \leq i < m$ (i.e. each of m possible values is equally likely for a random key)
- fast (constant time) computation
- different values even for very similar keys

Example:

- $h(k) = k \bmod m$ (usually m is a prime number)

Hashing always has to deal with **collisions** (when $h(k) == h(j)$ for two keys $k \neq j$)

Collisions

Assume a new key k comes on position $h(k)$ that is **not free**.

Two common ways of dealing with collisions in hash tables are:

- k is added to a *list* $l(h(k))$ kept at position $h(k)$:
(**chaining method**)
- other indexes are scanned (in a *repeatable* way) until a free index is found: (“**open hashing**”)

Chain Method

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

n - number of elements kept
compute $h(k)$: $O(1)$

- insert: compute $h(k)$ and add new element to the list at $h(k)$: $O(1)$
- find: compute $h(k)$ and scan the list $l(h(k))$ to return the element: $O(1 + |l(h(k))|)$
- delete: compute $h(k)$, scan $l(h(k))$ to remove the element: $O(1 + |l(h(k))|)$

Complexity depends on the length of list $l(h(k))$.

Note: worst case (for $|l(h(k))| == n$) needs $\Theta(n)$ comparisons (worst case is not better than in naive implementation!)

Average Case Analysis of Chain Method

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

If hash function satisfies *uniform load* assumption, chain method guarantees **average** of $O(1 + \alpha)$ comparisons for all dictionary operations, where $\alpha = n/m$ (load factor). Thus, if $m = O(n)$ chain methods results in average $O(1)$ time for all dictionary operations.

Proof: Assume a random key k to be hashed. Let X denote random variable representing the length of a list $I(h(k))$. Any operation needs constant time for computing $h(k)$ and then linearly scans the list $I(h(k))$, and thus costs $O(1 + E[X])$. Let S be the set of elements kept in hashtable, and for $e \in S$ let X_e denote *indicator* random variable such that $X_e == 1$ iff $h(k) == h(e)$ and 0 otherwise². We have $X = \sum_{e \in S} X_e$. Now,

$$E[X] = E\left[\sum_{e \in S} X_e\right] = \sum_{e \in S} E[X_e] = \sum_{e \in S} P(X_e == 1) = |S| \frac{1}{m} = \frac{n}{m}$$

Thus $O(1 + E[X]) = O(1 + \alpha)$.

²Can be denoted shortly as: $X_e = [h(k) == h(e)]$

Universal Hashing

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

Family H of hash functions into range $0, \dots, m - 1$ is called *c-universal*, for $c > 0$, if for randomly chosen hash function $h \in H$ any two distinct keys i, j collide with probability:

$$P(h(i) == h(j)) \leq c/m$$

Family H is called *universal* if $c == 1$

To avoid “malicious” data, hash function can be first randomly picked from a c -universal hashing family.

If c -universal hashing family is used in chain method, average time of dictionary operations is $O(1 + cn/m)$

Open Hashing

In open hashing, there is exactly one element on each position. Consider `insert` operation: if, for a new k , $h(k)$ is already in use, the entries are scanned in a specified (and repeatable) order $\pi(k) = (h(k, 0), h(k, 1), \dots, h(k, m - 1))$ until a free place is found. `find` is analogous, `delete` additionally needs to restore the hash table after removing the element.

- linear: $h(k, i) = (h(k) + i) \bmod m$
(problem: elements tend to group (“primary” clustering))
- quadratic: $h(k, i) = (h(k) + c_1 i + c_2 i^2) \bmod m$
(problem: “secondary” clustering: if the first positions are equal, all the other are still the same)
- re-hashing: $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$ (h_1, h_2 should differ, e.g.: $h_1(k) = k \bmod m$, $h_2(k) = 1 + (k \bmod m')$, $m' = m - 1$
(here, the order permutations are “more random”))

Average Case Analysis of Open Hashing

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

In open hashing, under assumption that all scan orders are equally probable, *find* have guaranteed **average** number of comparisons:

- $\frac{1}{1-\alpha}$ if the key to be found is absent
- $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$ if the key to be found is present

($\alpha = n/m < 1$ (load factor))

In open hashing, the **worst-case** number of comparisons is linear. In addition it is necessary that $n < m$. When n approaches m open hashing becomes as slow as on unordered linear sequence (naive implementation of dictionary).

(*) Perfect Hashing

Previous methods guarantee *expected* constant time of dictionary operations.

Perfect hashing is a scheme that guarantees *worst case* constant time.

It is possible to construct a perfect hashing function, for a given set of n elements to be hashed, in expected (i.e. average) linear time: $O(n)$

(the construction can be based on some family of 2 – *universal* hash functions (Fredman, Komlos, Szemeredi 1984))

Dynamic Ordered Set

Abstract data structure that is an extension of the dictionary:
(and we assume that type K is linearly ordered)

- `search(K key)`
- `insert(K key, V value)`
- `delete(K key)`
- `minimum()`
- `maximum()`
- `predecessor(K key)`
- `successor(K key)`

Hash table is a very good implementation of the first three operations (dictionary operations) however does not efficiently support the new four operations concerning the order of the keys.

Binary Search Tree

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

BST is a binary tree, where keys (contained in the tree nodes) satisfy the following condition (so called “BST order”):

For each node, the key contained in this node is higher or equal than all the keys contained in the left subtree of this node and lower or equal than all keys in its right subtree

Where is the minimum key? Where is the maximum key?

Search Operation

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

```
searchRecursive(node, key): \\ called with node == root
    if ((node == null) or (node.key == key)) return node
    if (key < node.key) return search(node.left, key)
    else return search(node.right, key)
```

```
searchIterative(node, key): \\ called with node == root
    while ((node != null) and (node.key != key))
        if (key < node.key) node = node.left
        else node = node.right
    return node
```

Minimum and Maximum

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

```
minimum(node): \\ called with node == root
    while (node.left != null) node = node.left
    return node
```

```
maximum(node): \\ called with node == root
    while (node.right != null) node = node.right
    return node
```

```
successor(node):
    if (node.right != null) return minimum(node.right)
    p = node.parent
    while ((p != null) and (node == p.right))
        node = p
        p = p.parent
    return p
```

(predecessor is analogous to successor)

Example insert Implementation

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

```
insert(node, key):
    if (key < node.key) then
        if node.left == null:
            n = create new node with key
            node.left = n
        else: insert(node.left, key)
    else: // (key >= node.key)
        if node.right == null:
            n = create new node with key
            node.right = n
        else: insert(node.right, key)
```


Example delete Implementation

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

```
procedure delete(node, key)
  if (key < node.key) then
    delete(node.left, key)
  else if (key > node.key) then
    delete(node.right, key)
  else begin { key = node.key
    if node is a leaf then
      deletesimple(node)
    else
      if (node.left != null) then
        find x = the rightmost node in node.left
        node.key:=x.key;
        delete1(x);
      else
        proceed analogously for node.right
        (we are looking for the leftmost node now)
```

Example of a helper delete1 Implementation

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

```
// delete1: for nodes having only 1 son

procedure delete1(node)
begin
  subtree = null
  parent = node.parent
  if (node.left != null)
    subtree = node.left
  else
    subtree = node.right

  if (parent == null)
    root = subtree
  else if (parent.left == node) // node is a left son
    parent.left = subtree
  else // node is a right son
    parent.right = subtree
```

BST: Average Case Analysis

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary


For simplicity assume that keys are unique.

Assume that every permutation of n elements inserted to BST is equally likely³ it can be proved that average height of BST is $O(\log n)$.

Two cases for operations concerning a key k :

- k is not present in BST: in this case the complexities are bounded by **average height** of a BST
- k is present in BST: in this case the complexities of operations are bounded by **average depth** of a node in BST

An expected height of a random-permutation model BST can be proved to be $O(\log n)$ by analogy to QuickSort (the proof is omitted in this lecture)

³If we assume other model: i.e. that every n -element BST is equally likely, the average height is $\Theta(\sqrt{n})$. This model seems to be less natural, 

(*) Average Depth of a Node in BST (random permutation model)

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

We will explain that the average depth is $O(\log n)$ (formal proof is omitted but it can be easily derived from the explanation)

For a sequence of keys $\langle k_i \rangle$ inserted to a BST define:

$G_j = \{k_i : 1 \leq i < j \text{ and } k_i > k_j \text{ for all } l < i \text{ such that } k_l > k_j\}$

$L_j = \{k_i : 1 \leq i < j \text{ and } k_l < k_i < k_j \text{ for all } l < i \text{ such that } k_l < k_j\}$

Observe, that the path from root to k_j consists exactly from $G_j \cup L_j$ so that the depth of k_j will be $d(k_j) = |G_j| + |L_j|$

G_j consists of the keys that arrived before k_j and are its direct successors (in current subsequence). The i -th element in a random permutation is a current minimum with probability $1/i$. So that the expected number of updating minimum in n -element random permutation is $\sum_{i=1}^n 1/i = H_n = O(\log n)$. Being a current minimum is necessary for being a direct successor. Analogous explanations hold for L_j . So that the upper bound holds: $d(k_j) = O(\log n)$.

BST: Complexities of Operations

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

data size: number of elements in dictionary (n)
dominating operation: comparison of keys

Average time complexities on BST are:

- search $\Theta(\log n)$
- insert $\Theta(\log n)$
- delete $\Theta(\log n)$
- minimum/maximum $\Theta(\log n)$
- successor/predecessor $\Theta(\log n)$

The worst-case complexities of operations on BST is $O(n)$.

AVL tree (Adelson-Velskij, Landis)

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

AVL is the simplest tree data structure for ordered dynamic dictionary to guarantee $O(\log n)$ worst-case height.

AVL is defined as follows:

AVL is a BST with the **additional** condition: for each node the difference of height of its left and right sub-tree is not greater than 1.

Maximum Height of an AVL Tree

Let T_h be a minimum number of nodes in an AVL tree that has height h .

Observe that:

- $T_0 = 1, T_1 = 2$
- $T_h = 1 + T_{h-1} + T_{h-2}$
(consider left and right subtrees of the root)

Thus: $T_h \geq F_h$ (Fibonacci number). Remind: h -th Fibonacci number has exponential growth (in h). Since the minimum number of nodes in AVL has at least exponential growth in height of the tree (h), the height of AVL has at most **logarithmic** growth in the number of nodes.

Thus, the height of n -element AVL tree has worst-case guarantee of $O(\log n)$.

Implementation of operations on AVL

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

The same as on BST **but**:

- with each node a **balance factor** (bf) is kept (= the difference in heights between left and right subtree of the given node)
- after each operation, bf is updated for each affected node
- if, after a modifying operation, the value of bf is outside of the set of values $\{-1, 0, 1\}$ for some nodes - the **rotation** operations are called (on these nodes) to re-balance the tree.

AVL Rotations

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

All the dictionary operations on AVL begin in the same way as in the BST. However, after each modifying operation on this tree the bf values are re-computed (bottom-up)

Moreover, if after any modifying operation any bf is 2 or -2, a special additional operation called **rotation** is executed for the node.

There are 2 kinds of AVL rotations: single and double and both have 2 mirror variants: left and right.

Each rotation has $O(1)$ time complexity.

The rotations are defined so that the height of the subtree rooted at the “rotated” node is preserved. Why is it important? (among others) due to this $|bf|$ cannot exceed 2 after any operation/rotation on a valid AVL tree.

AVL: Worst-case Analysis of Operations

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST


Summary

To summarise:

- each rotation has $O(1)$ complexity
- (as in BST) the complexities of operations are bounded by the height of the tree
- an n -element AVL tree has **at most logarithmic height**

Thus: all dictionary operations have guaranteed $O(\log n)$ worst-case complexities on AVL.

Note: the maximum number of rotations after a single delete operation could be logarithmic on n , though. ⁴

⁴This may happen on a Fibonacci tree. To see example: Donald Knuth, "The Art of Computer Programming", vol. 3: "Sorting" 

Self-organising BST (or Splay-trees)

Guarantee amortised $O(\log n)$ complexity for all ordered dictionary operations. More precisely, any sequence of m operations will have total complexity of $O(m \log n)$.

Idea: each operation is implemented with a helper **splay(k)** operation, where k is a key:

- **splay(k)**: by a sequence of rotations bring to the root either k (if it is present in the tree) or its direct successor or predecessor
- **insert(k)**: **splay(k)** (to bring successor (predecessor) k' of k to the root), then make k' the right (left) son of k
- **delete(k)**: **splay(k)** (k becomes the root), remove k (to obtain two separate subtrees), **splay(k)** again on the left (right) subtree (to bring predecessor (successor) k' of k to the root), make k' of the right (left) orphaned subtree.

It can be proved that the insert and delete operations (described above) have amortised logarithmic time complexities.

Large on-disk dictionaries

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

There are special data structures designed for implementing dictionary in case it does not fit to memory (mostly kept on disk).

Example: B-trees (and variants). The key idea: minimize the disk read/write activity (node should fit in a single disk block size)

Used in DB implementations (among others).

Dictionaries Implementations: Brief Summary of the Lecture

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary
Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

- Hashtables provide very fast operations but do not support ordering-based operations (as successor, minimum, etc.)
- BST is the simplest implementation of ordered dictionary that guarantees average logarithmic complexities, but have linear pessimistic complexities
- AVL is an extension of BST that guarantees even worst-case logarithmic complexities through rotations. Additional memory is needed for *bf*
- self-organising BST also guarantees worst-case logarithmic complexities through splay operation (based on rotations), without any additional memory (compared to BST). Interesting property: automatic adaptation to non-uniform access frequencies.
- B-trees, AB-trees, B^+ -trees, etc. - large, on-disk structures

Questions/Problems:

Algorithms
and Data
Structures

Marcin
Sydow

Dictionary

Hashtables

Dynamic
Ordered Set

BST

AVL

Self-
organising
BST

Summary

- Dictionary
- Hashing
 - Chain Method
 - Open Hashing
 - Universal Hashing
 - Perfect Hashing
- Ordered Dynamic Set
- BST
- AVL
- Self-organising BST
- Comparison of different implementations

Thank you for attention