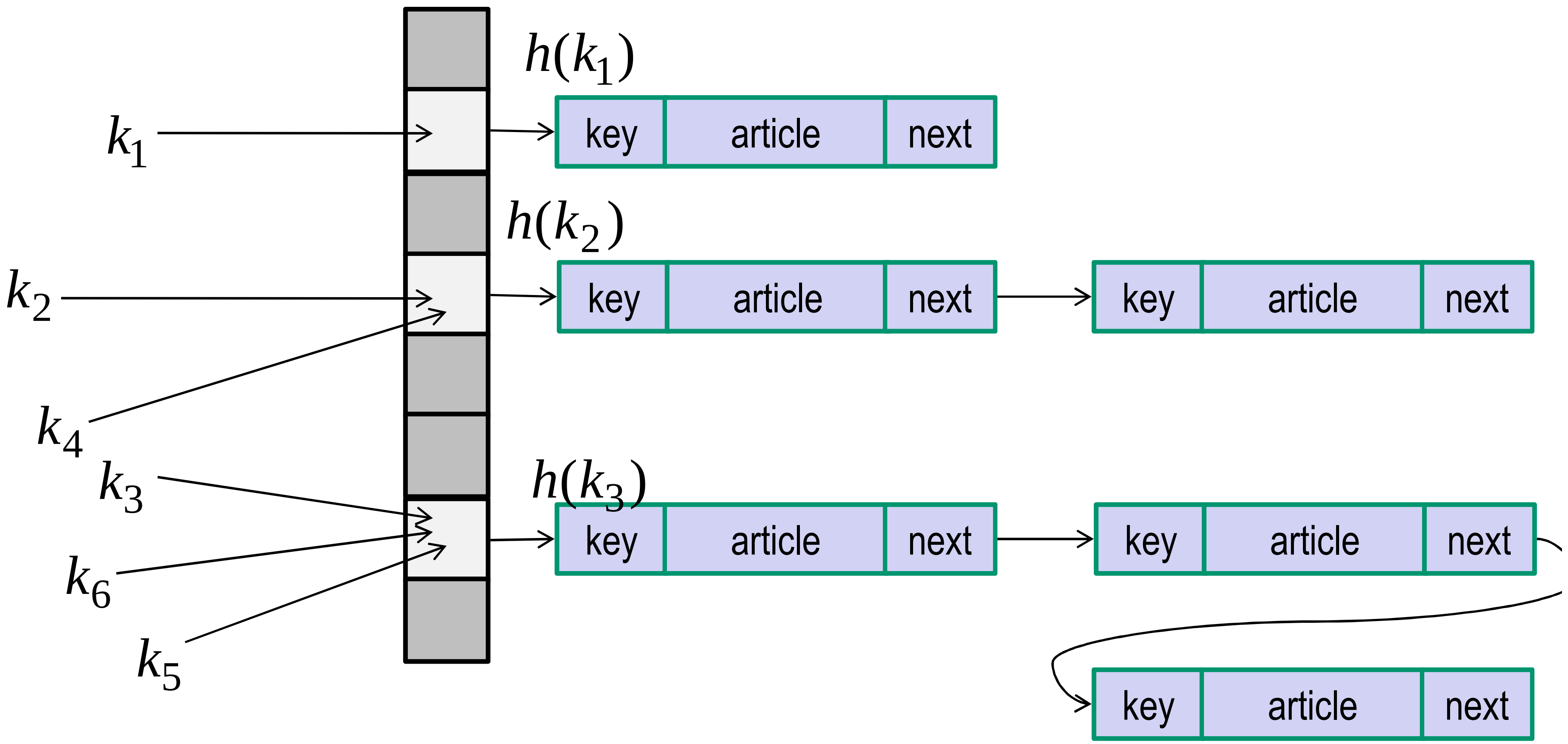
Hash Tables II

Data Structures and Algorithms

Andrei Bulatov

Hash Tables

In case of collision create a list of elements with the same hash value



Good Hash Functions

Good hash functions are those that are as close to simple uniform hashing as possible

It is difficult to achieve, since we do not know the distribution of keys

Note, there are two types of hash functions with absolutely different requirements:

- hash functions to support data structures
- cryptographic hash functions

Assumption:

All keys are natural numbers

The Division Method

Choose m

Then $h(k) = k \bmod m$

Should be careful with some values of m

Say, no powers of 2, or powers of 10, or …

Primes is a good choice, as long as they are not close to a power of 2

The Multiplication Method

Choose m

Choose A with $0 < A < 1$

x mod 1 denotes the fractional part of x, that is $x - \lfloor x \rfloor$

Then $h(k) = \lfloor m\,(kA \bmod 1) \rfloor$

$m = 2^p$ is a convenient value

If the size of a computer word is w, choose A to be a fraction like $\frac{s}{2^w}$ for a integer s

To compute h(k), multiply k by $s = A \cdot 2^w$

The result is a 2w-bit value $r_1 2^w + r_0$

Then h(k) is then the p most significant bits of r_0

Universal Hashing

To guarantee hashing even closer to simple uniform, a natural idea is to choose hash function also at random, independent of the keys being hashed

We use universal collection of hash functions

A collection H of hash functions is called universal, if for each pair of distinct keys k and l, the number of hash functions $h \in H$ such that $h(k) = h(l)$ is no more than $|H|/m$

To construct a hash table we first select $h \in H$ (randomly!), and then use it

Universal Hashing (cntd)

Lemma

Suppose a hash function is chosen at random from a universal collection and is used to hash n keys into a table of size m.

If key k is not in the table, then the expected length $E[n_{h(k)}]$ of the list that k hashes to is at most α = n/m.

If k is in the table, then the expected length $E[n_{h(k)}]$ of the list containing k is at most 1 + α

Corollary

Using universal hashing and collision resolution by chaining in a table with m slots, it takes expected time Θ(n) to handle any sequence of n table operations.

Constructing a Universal Hashing Collection

Choose a prime p such that all possible keys are in the range {0, …, p – 1}

Let $Z_p = \{0, \ldots, p-1\}$ and $Z_p^* = \{1, \ldots, p-1\}$

For $a \in Z_p^*$ and $b \in Z_p$ let

$$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$$

and $H_{p,m} = \{h_{a,b} : a \in Z_p^*, b \in Z_p\}$

Theorem

The class $H_{p,m}$ of hash functions is universal

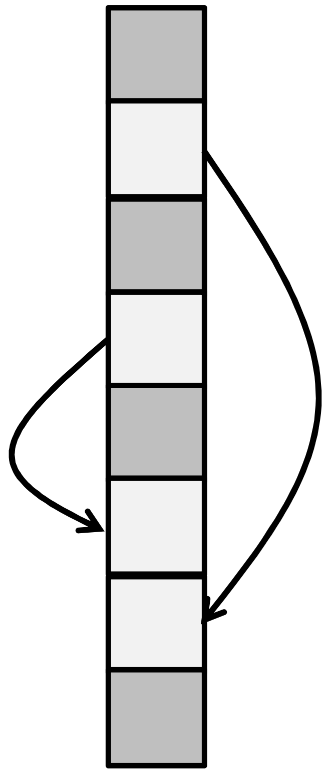
Open Addressing

A serious drawback of chaining: it uses a lot of pointers

The idea:

Keep all the lists inside the hash table

Instead of using pointers, compute the location of the next element



To insert or search the hash table we successfully check or probe a sequence of entries of the table

This sequence depends on the key being searched or inserted

Probe Sequence

Hash function depends on 2 arguments and generates a probe sequence

Formally:

$h: U \times \{0,1, \ldots, m-1\} \to \{0,1, \ldots, m-1\}$

Probe sequence

$\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$

We want this sequence to be a permutation of $0,1, \ldots, m-1$, so that every slot in the hash table can be occupied.

Clearly we cannot store more elements than the number of slots in the table

Thus the load factor does not exceed 1

Insertion

Hash-Insert(T,k)

```
set i:=0
repeat
  set j:=h(k,i)
  if T[j]=Nil then do
     set T[j]:=k
     return j
  else  set i:=i+1
until i=m
error “hash table overflow”
```

Search and Deletion

Hash-Search(T,k)

```
set i:=0
repeat
  set j:=h(k,i)
  if T[j]=k then return j
  set i:=i+1
until T[j]=Nnil or i=m
return Nil
```

Deletion is difficult, as it is not possible in general to shift all elements in a sequence, for some of them may belong to different sequences

We can write `Deleted' instead of actual deleting

Or better use chaining

Probing: Linear

To generate a probe sequence we use an ordinary hash function, called auxiliary hash function

$h': U \to \{0,1, \ldots, m-1\}$

Linear probing:

$h(k,i) = (h'(k) + i) \bmod m$

Thus we start searching from slot $h'(k)$, then check $h'(k) + 1$, etc.

Drawbacks:

- Primary clustering, long sequences of occupied slots build up making the average search time too long
- Since $h(k,0) = h(k',0)$ implies $h(k,i) = h(k',i)$ for all i, there are very few different probe sequences (m to be precise)

Probing: Quadratic

Quadratic probing:

$h(k,i) = (h'(k) + c \cdot i + d \cdot i^2) \bmod m$

where h' is an auxiliary hash function, $c,d \neq 0$ are constants

No primary clustering

Drawbacks:

- Possible values of c, d, and m are very restricted
- Secondary clustering, milder form of clustering
- Only few different probe sequences

Probing: Double Hashing

Double hashing uses two auxiliary hash functions

$h(k,i) = (h'(k) + i \cdot h''(k)) \bmod m$

where h' and h'' are an auxiliary hash functions

Thus the sequence depends on the value of two hash functions

It is unlikely it produces any kind of clustering

Also if h' and h'' are selected properly, we have m^2 different probe sequences

Choice of h' and h'':

- $h''(k)$ should be relatively prime to m to make sure we search the entire table
- Say, m is a power of 2, and $h''(k)$ is always odd
- Or m is prime, and $h''(k) < m$ for all k
  - $h'(k) = k \bmod m$
  - $h''(k) = 1 + (k \bmod m')$, and $m' = m - 1$

Open Addressing Analysis

Theorem

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$ assuming uniform hashing

Theorem

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

assuming uniform hashing

Homework

Suggest how to organize a direct access table in which not all keys are different. All operations must run in $O(1)$ time

Show that if $|U| > mn$ (U denotes the set of all possible keys), there is a subset of U of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$

Binary Search Trees

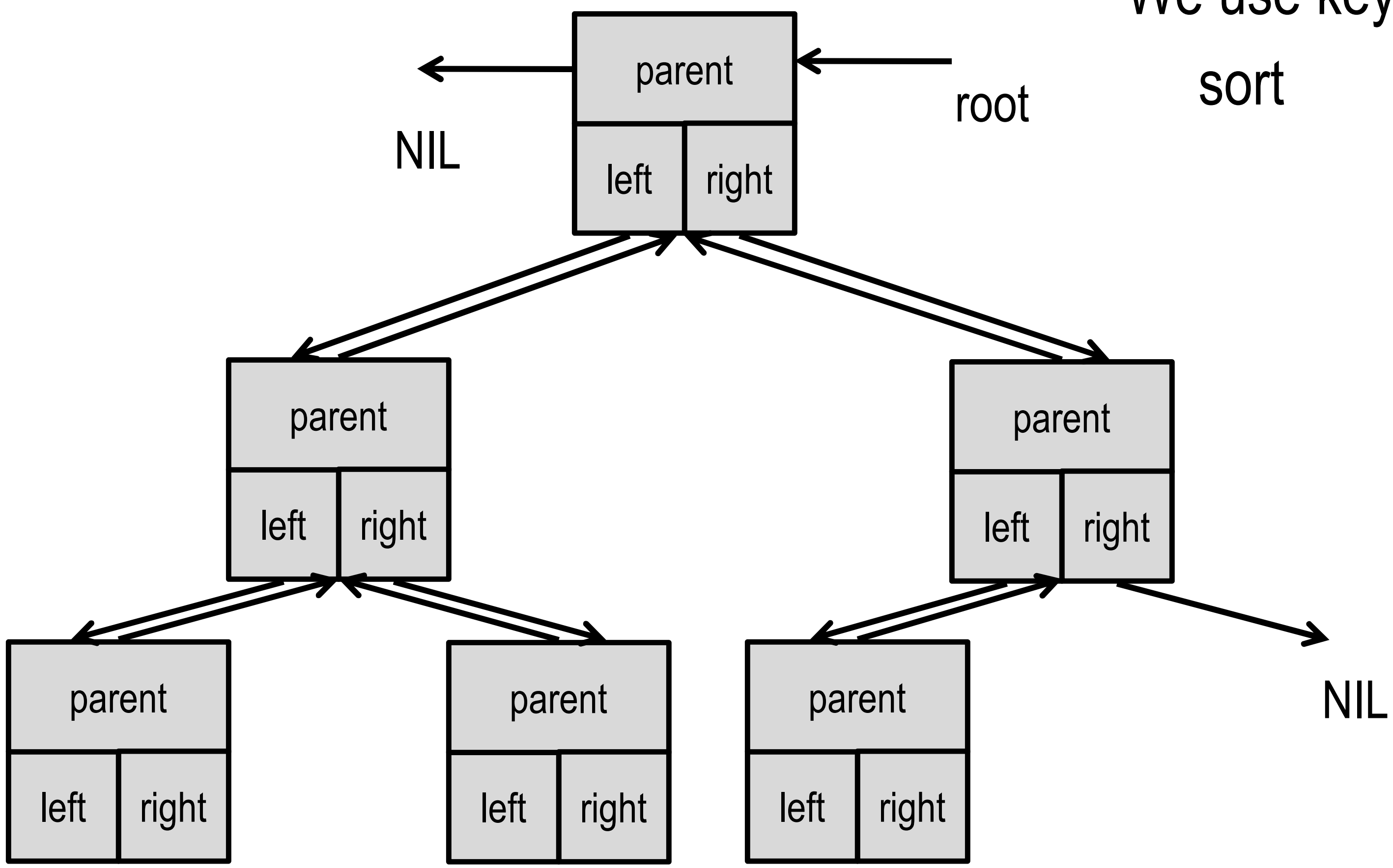
Data Structures and Algorithms

Andrei Bulatov

Binary Rooted Trees

Another good way to store dictionaries and other sorted data

A binary tree is used



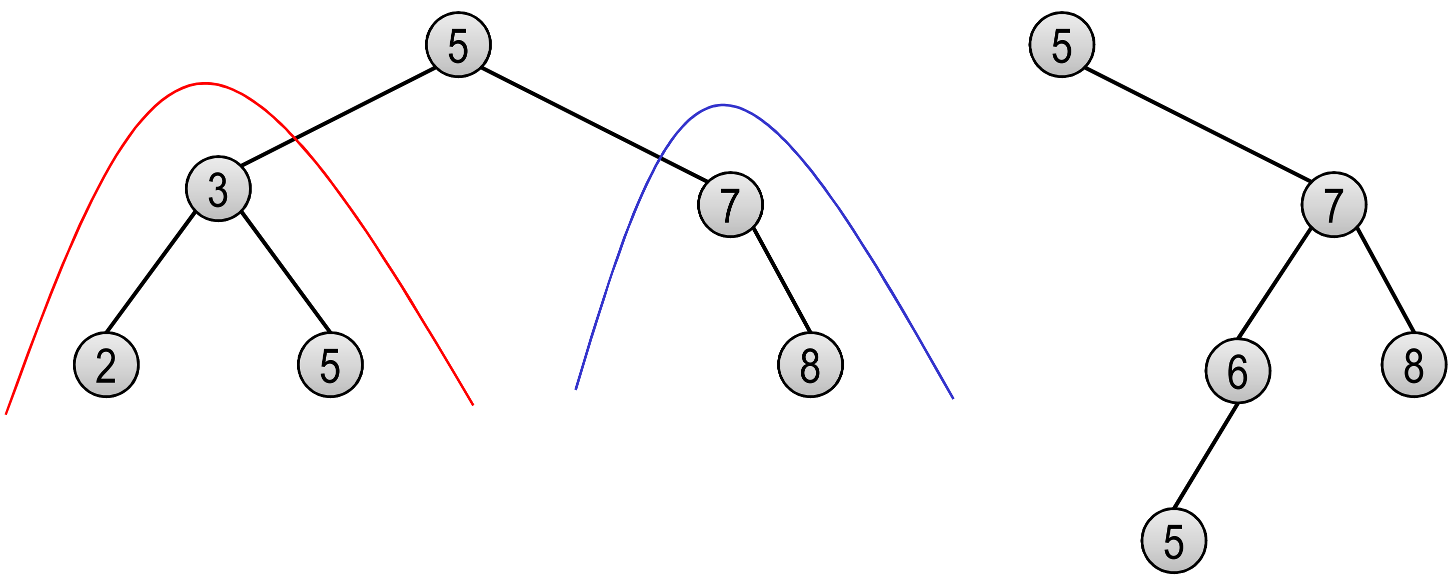
We use key to sort

Binary Search Tree

A binary search tree is a binary rooted tree, in which keys satisfy the Binary Search Tree property:

Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key[y] ≤ key[x].

If y is a node in the right subtree of x, then key[x] ≤ key[y]



Inorder Tree Walk

Having a binary search tree one can print its content in sorted order

Inorder-Tree-Walk(x)

```
if x=Nil then do
     Inorder-Tree-Walk(left[x])
     print key[x]
     Inorder-Tree-Walk(right[x])
```

To print the entire tree just call Inorder-Tree-Walk(root[T])

It is called inorder because the root is printed between the subtrees

A tree walk can also be preorder and postorder

Inorder Tree Walk (cntd)

Lemma

If x is the root of an n-node subtree, then the call Inorder-Tree-Walk(x) takes $\Theta(n)$ time

Proof

Let $T(n)$ denote the running time

If $x = Nil$ then $T(0) = c$, a constant

If $n > 0$ then $T(n) = T(k) + T(n - k - 1) + d$ where k is the number of nodes in the left subtree

We prove that $T(n) = (c+d)\, n + c$

For $n = 0$ we have $(c + d) \cdot 0 + c = c = T(0)$

Inorder Tree Walk (cntd)

Proof

For $n > 0$ we have

$$
\begin{aligned}
T(n) &= T(k) + T(n-k-1) + d \\
&= ((c+d)k + c) + ((c+d)(n-k-1) + c) + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= (c+d)n + c
\end{aligned}
$$

QED

Searching

Elements of a binary search tree can be found efficiently

Tree-Search(x,k)

```
if x=Nil or k=key[x] then
   return x
if  k<key[x] then
   return Tree-Search(left[x],k)
else
   return Tree-Search(right[x],k)
```

Minimum and Maximum

We can find minimum and maximum keys in the tree

Tree-Minimum(x)

```
while left[x]≠Nil do
   set x:=left[x]
endwhile
return x
```

Tree-Maximum(x)

```
while right[x]≠Nil do
   set x:=right[x]
endwhile
return x
```

Successor and Predecessor

Sometimes we need to find the successor or predecessor of

Tree-Successor(x)

```
if right[x]≠Nil then
   return Tree-Minimum(right[x])
set y:=paret[x]
while y≠Nil and x=right[y] do
   set x:=y
   set y:=parent[y]
endwhile
return y
```

Running Time of Tree Operations

Theorem

The operations Search, Minimum, Maximum, Successor, and Predecessor can be made to run in $O(h)$ time on a binary search tree of height h