Data Structures 1

NTIN066

Jiří Fink

https://kam.mff.cuni.cz/~fink/

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Winter semester 2015/16
Last change on February 3, 2016

Content

Jiří Fink: Data Structures 1

Plan of the lecture

- Trees
  - (a,b)-trees
  - MFR-strategy for lists, Splay trees
  - Other solutions: AVL trees, red-black trees, BB-alpha trees
- Heaps
  - Regular heaps
  - Binomial heaps - amortized and worst-case complexity
  - Fibonacci heaps
- Techniques for memory hierarchy
  - I/O model, cache-oblivious analysis, LRU-strategy for on-line paging
  - Examples: matrix transposition and multiplication, van Emde Boas tree layout
- Hashing
  - Collisions and their resolution, analysis for uniformly distributed data
  - Selecting a hash function: universal hashing, good hash functions
  - Cuckoo hashing
- Multidimensional data structures
  - KD trees
  - Range trees

Jiří Fink: Data Structures 1

General information

E-mail	fink@kam.mff.cuni.cz
Homepage	http://kam.mff.cuni.cz/~fink/
Consultations	Individual schedule

Examination

- Successfully work out four out of five homeworks
- Pass the exam

Jiří Fink: Data Structures 1

Literature

- A. Koubková, V. Koubek: Datové struktury I. MATFYZPRESS, Praha 2011.
- T. H. Cormen, C.E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms. MIT Press, 2009
- K. Mehlhorn: Data Structures and Algorithms I: Sorting and Searching. Springer-Verlag, Berlin, 1984
- D. P. Mehta, S. Sahni eds.: Handbook of Data Structures and Applications. Chapman & Hall/CRC, Computer and Information Series, 2005
- E. Demaine: Cache-Oblivious Algorithms and Data Structures. 2002.
- R. Pagh: Cuckoo Hashing for Undergraduates. Lecture note, 2006.
- M. Thorup: High Speed Hashing for Integers and Strings. lecture notes, 2014.
- M. Thorup: String hashing for linear probing (Sections 5.1-5.4). In Proc. 20th SODA, 655-664, 2009.

Outline

Dictionary problem

Entity

- Entity is a pair of a key and a value
- Keys are linearly ordered
- Number of entities stored in a data structure is n

Basic operations

- Insert a given entity
- Find an entity of a given key
- Delete an entity of a given key

Example of data structures

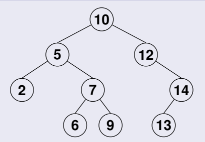
- Array
- Linked list
- Searching trees (e.g. AVL, red-black)
- Hash tables

Binary search tree

Properties

- Entities are stored in nodes (vertices) of a rooted tree
- Each node contains a key and two sub-trees (children), the left and the right
- The key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree

Example



Complexity

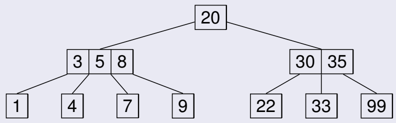
- Space: $\mathcal{O}(n)$
- Time: Linear in the depth of the tree
- Height in the worst case: n

(a,b)-tree

Properties

- a, b are integers such that $a \geq 2$ and $b \geq 2a - 1$
- All internal nodes except the root have at least a and at most b children
- The root has at most b children
- All leaves are at the same depth
- Entities are stored in leaves
- Keys in leaves are ordered (left-to-right)
- Internal nodes contain keys used to find leaves (e.g. the largest key)

Example: (2,4)-tree

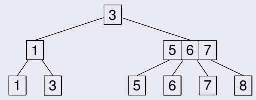


Operation Find

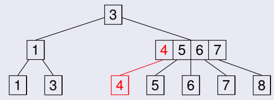
Search from the root using keys stored in internal nodes

(a,b)-tree: Insert

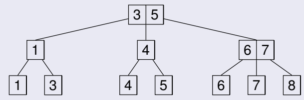
Insert 4 into the following (2,4)-tree



Add a new leaf into the proper parent



Recursively split node if needed



(a,b)-tree: Insert

Algorithm

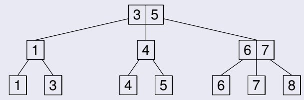
Find the proper parent v of the inserted entity
Add a new leaf into v
while deg(v) $> b$ **do**
    `# Find parent` u `of node` v
    if *v is the root* **then**
        Create a new root with v as its only child
    else
        $u \leftarrow$ parent of v
    `# Split node` v `into` v `and` v'
    Create a new child v' of u immediately to the right of v
    Move the rightmost $\lfloor (b+1)/2 \rfloor$ children of v to v'
    $v \leftarrow u$

Time complexity

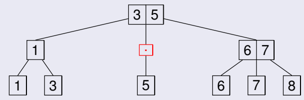
Linear in height of the tree

(a,b)-tree: Delete

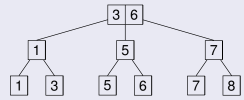
Delete 4 from the following (2,4)-tree

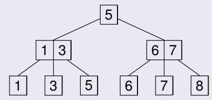


Find and delete the proper leaf



Recursively either share nodes from a sibling or fuse the parent





(a,b)-tree: Delete

Algorithm

Find the leaf l containing the deleted key
$v \leftarrow$ parent of l
Delete l
while deg(v) $< a$ *and v is not the root* **do**
    $u \leftarrow$ an adjacent sibling of v
    if deg(u) $> a$ **then**
        Move the proper child from u to v
    else
        Move all children of u to v
        Remove u
        if *If v has no sibling* **then**
            Remove the root (= parent of v) and make v the new root
        else
            $v \leftarrow$ parent of v

(a,b)-tree: Analysis

Height

- (a,b)-tree of height d has at least a^{d-1} and at most b^d leaves.
- Height satisfies $\log_b n \leq d \leq 1 + \log_a n$.

Complexity

The time complexity of operations find, insert and delete is $\mathcal{O}(\log n)$.

(a,b)-tree: Join

Description

Union of two (a,b)-trees T_1 and T_2 assuming max key(T_1) $<$ min key(T_2).

Algorithm

if *height*(T_1) $\geq$ *height*(T_2) **then**
    $u \leftarrow$ last node of T_1 in height *height*(T_1) $-$ *height*(T_2)
    $v \leftarrow$ root of T_2
else
    $u \leftarrow$ last node of T_2 in height *height*(T_2) $-$ *height*(T_1)
    $v \leftarrow$ root of T_1
Move all children of v to u
if deg(u) $> b$ **then**
    Recursively split u like in the operation insert

Complexity

Linear in the difference of heights of trees.

(a,b)-tree: Split

Description

Given an (a,b)-tree T and a key k, split T to two (a,b)-trees T_S and T_G with keys smaller and greater than k, respectively.

Algorithm (only for T_S)

Input: (a,b)-tree T, key x
$Q_S \leftarrow$ an empty stack
$t \leftarrow$ the root of T
while *t is not a leaf* **do**
    $v \leftarrow$ child of t according to the key x
    Push all left brothers of v to Q_S
    $t \leftarrow$ v
$T_S \leftarrow$ an empty (a,b)-tree
while *Q_S is non-empty* **do**
    $T_S \leftarrow$ JOIN(POP(Q_S), T_S)

Time complexity

$\mathcal{O}(\log n)$ since complexity of JOIN is linear in the difference of heights of trees.

(a,b)-tree: Ord

Description
Returns the i-th smallest key in the tree for given i.

Approach
If every node stores the number of leaves in its sub-tree, the i-th smallest key can be fount in $\mathcal{O}(\log n)$-time.

Note
Updating the number of leaves does not influence the time complexity of operations insert and delete.

(a,b)-tree: The amortized number of balancing operations

Assumption
$b \geq 2a$

Statement (without proof)
The number of balancing operations for l inserts and k deletes is $\mathcal{O}(l + k + \log n)$.

Conclusion
The amortized number of balancing operations for one insert or delete is $\mathcal{O}(1)$.

(a,b)-tree for parallel access

Assumption
$b \geq 2a$

Operation insert
Split every node with b children on path from the root to the inserted leaf.

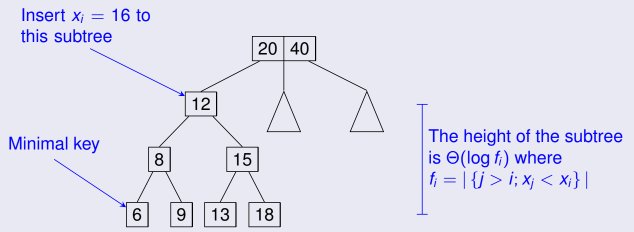
Operation delete
Update (move a child or merge with a sibling) every node with a children on path from the root to the deleted leaf.

A-sort

Goal
Sort "almost sorted" list $x_1, x_2, \ldots, x_n$.

Modification of (a,b)-tree
The (a,b)-tree also stores the pointer to the most-left leaf.

Idea: Insert $x_i = 16$



A-sort: Algorithm

Input: list $x_1, x_2, \ldots, x_n$
$T \leftarrow$ an empty (a,b)-tree
for $i \leftarrow n$ **to** 1 **do**
  `# Modified operation insert of` x_i `to` T
  $v \leftarrow$ the leaf with the smallest key
  while *x_i is greater than the maximal key in the sub-tree of v and v is not a root* **do**
    $v \leftarrow$ parent of v
  Insert x_i but start searching for the proper parent at v
Output: Walk through whole (a,b)-tree T and print all leaves

A-sort: Complexity

The inequality of arithmetic and geometric means
If $a_1, \ldots, a_n$ are non-negative real numbers, then

$$\frac{\sum_{i=1}^n a_i}{n} \geq \sqrt[n]{\prod_{i=1}^n a_i}.$$

Time complexity
- Denote $f_i = |\{j > i;\, x_j < x_i\}|$
- $F = \sum_{i=1}^n f_i$ is the number of inversions
- Finding the starting vertex v for one key x_i: $\mathcal{O}(\log f_i)$
- Finding starting vertices for all keys: $\mathcal{O}(n \log(F/n))$
- Splitting nodes during all operations insert: $\mathcal{O}(n)$
- Total time complexity: $\mathcal{O}(n + n\log(F/n))$
- Worst case complexity: $\mathcal{O}(n \log n)$ since $F \leq \binom{n}{2}$
- If $F \leq n \log n$, then the complexity is $\mathcal{O}(n \log \log n)$

The complexity of finding the starting point follows from
$\sum_i \log f_i = \log \prod_i f_i = n \log \sqrt[n]{\prod_i f_i} \leq n \log \frac{\sum_i f_i}{n} = n \log \frac{F}{n}$.

(a,b)-tree: Applications

Similar data structures
- B-tree, B+ tree, B* tree
- 2-4-tree, 2-3-4-tree, etc.

Applicatins
- A-sort
- File systems e.g. Ext4, NTFS, HFS+, FAT
- Databases

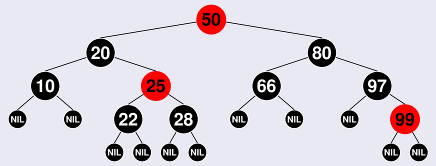
Outline

Red-black tree: Definition

Definition

- Binary search tree with elements stored in inner nodes
- Every inner node has two children — inner nodes or NIL/NULL pointers
- A node is either red or black
- Paths from the root to all leaves contain the same number of black nodes
- If a node is red, then both its children are black
- Leaves are black

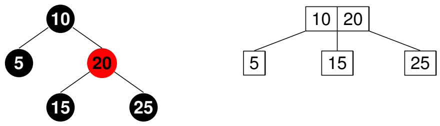
Example



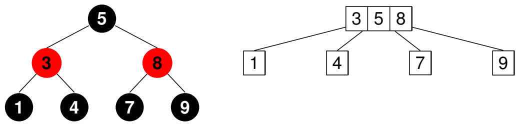
Red-black tree: Equivalence to (2,4)-tree

- A node with no red child

- A node with one red child



- A node with two red children



Red-black tree: Properties

Equivalence to (2,4)-tree

- Recolour the root to be black
- Combine every red node with its parent

Height

Height of a red-black tree is $\Theta(\log n)$

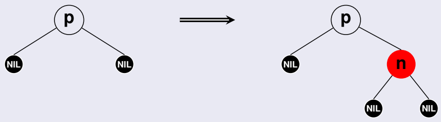
Applications

- Associative array e.g. std::map and std::set in C++, TreeMap in Java
- The Completely Fair Scheduler in the Linux kernel
- Computational Geometry Data structures

Red-black tree: Insert

Creating new node

- Find the position (NIL) of the new element n
- Add a new node



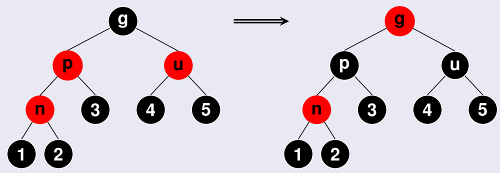
- If the parent p is red, balance the tree

When balancing

- A node n and its parent p are red. Every other property is satisfied.
- The grandparent g is black.
- The uncle u is red or black.

Red-black tree: Insert — uncle is red

Schema

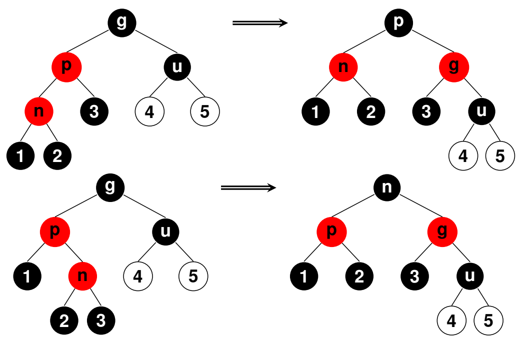


Notes

- In the equivalent (2,4)-tree, node g has five children (1,2,3,4,5).
- We "split" the node g by recolouring.
- If the great-grandparent is red, the balancing continues.

Red-black tree: Insert — uncle is black

In the equivalent (2,4)-tree, node g has four children (1,2,3,u).
The last balancing operation has two cases.



Outline

Amortized analysis

Amortized analysis

In an amortized analysis, the time required to perform a sequence of a data-structure operations is averaged over all the operations performed. The common examples are

	Worst-case	amortized
Incrementing a binary counter n	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Insert into a dynamic array	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Multi-pop from a stack	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Methods

- Aggregate analysis
- Accounting method
- Potential method

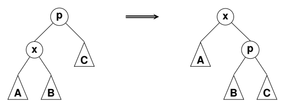
Splay tree

Description

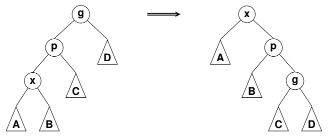
- Binary search tree
- Elements are stored in all nodes (both internal and leaves)
- Recently accessed elements are quick to access again
- Operation **splay** moves a given node to the root

Splay tree: splay a given node x

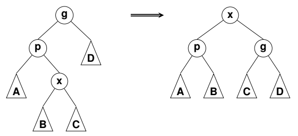
- Zig step: If the parent p of x is the root



- Zig-zig step: x and p are either both right children or are both left children



- Zig-zag step: x is a right child and p is a left child or vice versa



Splay tree: Amortized time

Lemma

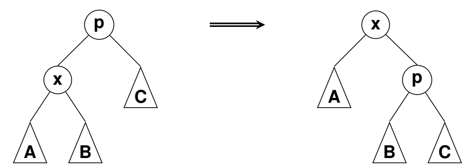
If $a + b \leq 1$, then $\log_2(a) + \log_2(b) \leq -2$.

Notations

- Size $s(x)$ is the number of nodes in the sub-tree rooted at node x (including x)
- Rank $r(x) = \log_2(s(x))$
- Potential Φ is the sum of the ranks of all the nodes in the tree
- s' and r' are size and rank functions after a splay step
- $\Delta\Phi$ is the change in the potential caused by a splay step

Since $4ab = (a+b)^2 - (a-b)^2$ and $(a-b)^2 \geq 0$ and $a + b \leq 1$, it follows that $4ab \leq 1$. Taking the logarithm of both sides, we derive $\log_2 4 + \log_2 a + \log_2 b \leq 0$, so the lemma holds.

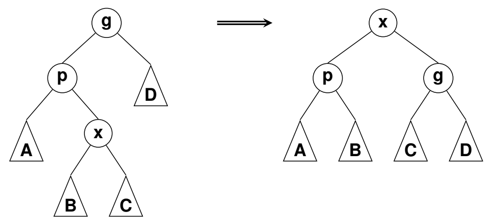
Splay tree: Zig step



Observe

- $r'(x) = r(p)$
- $r'(p) < r'(x)$
- $\Delta\Phi = \sum_x r'(x) - \sum_x r(x) = r'(p) - r(p) + r'(x) - r(x) \leq r'(x) - r(x)$

Splay tree: Zig-zag step



Observe

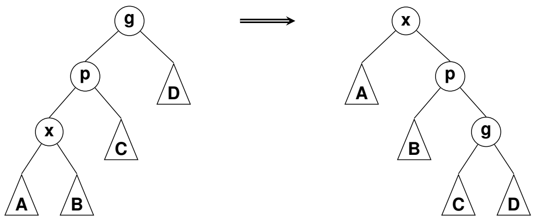
- $r'(x) = r(g)$
- $r(x) < r(p)$
- $s'(p) + s'(g) \leq s'(x)$
- $r'(p) + r'(g) \leq 2r'(x) - 2$
- $\Delta\Phi = r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \leq 2(r'(x) - r(x)) - 2$

From the third point follows $\frac{s'(p)}{s'(x)} + \frac{s'(g)}{s'(x)} \leq 1$, so we use the lemma to obtain

$$\begin{aligned}\log_2 \frac{s'(p)}{s'(x)} + \log_2 \frac{s'(g)}{s'(x)} &\leq -2\\ \log_2 s'(p) + \log_2 s'(g) &\leq 2\log s'(x) - 2.\end{aligned}$$

Now, we replace $\log s'(.)$ by the rank function $r'(.)$ to derive the fourth point.

Splay tree: Zig-zig step



Observe

- $r'(x) = r(g)$
- $r(x) < r(p)$
- $r'(x) > r'(p)$
- $s(x) + s'(g) \leq s'(x)$
- $r(x) + r'(g) \leq 2r'(x) - 2$
- $\Delta\Phi = r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \leq 3(r'(x) - r(x)) - 2$

Splay tree: Analysis

The amortized time

- The amortized cost of one zig-zig or zig-zag step:
  $\text{cost} + \Delta\Phi \leq 2 + 3(r'(x) - r(x)) - 2 = 3(r'(x) - r(x))$
- The amortized cost of one zig step:
  $\text{cost} + \Delta\Phi \leq 1 + 3(r'(x) - r(x))$
- The amortized time of whole operation splay:
  $\sum_{\text{steps}}(\text{cost} + \Delta\Phi) \leq 1 + 3(r(\text{root}) - r(x)) \leq 1 + 3\log_2 n = \mathcal{O}(\log n)$
- The amortized time for a sequence of m operations:
  $\mathcal{O}(m \log n)$
- The decrease in potential from the initial state Φ_i to the final state Φ_f:
  $\Phi_i - \Phi_f = \mathcal{O}(n \log n)$ since $0 \leq \Phi \leq n \log_2 n$

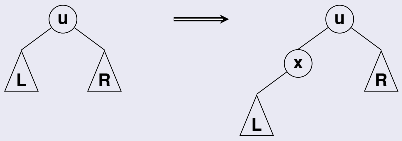
The actual time

The actual time for a sequence of m operations is $\mathcal{O}((n + m) \log n)$.

Splay tree: Insert

Insert key x

1. Find a node u with the closest key to x
2. Splay the node u
3. Insert a new node with key x



Amortized complexity

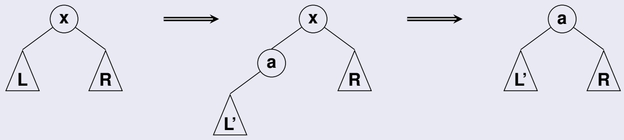
- Find and splay: $\mathcal{O}(\log n)$
- The potential Φ is increased by at most $r(x) + r(u) \leq 2 \log n$

Splay tree: Delete

Algorithm

```
Find and splay x
L ← the left subtree of x
if L is empty then
    Remove node x
else
    Find and splay the largest key a in L
    L′ ← the left subtree of a
    # a have no right child now
    Merge nodes x and a
```

L is non-empty



Outline

Heap

Basic operations

- Insert
- Find min
- Delete min
- Decrease key

Properties

- Entities are stored in all nodes of a tree
- The key of every node is always smaller than or equal to keys of its children

Applications

- Priority queue
- Heapsort
- Dijkstra's algorithm (find the shortest path between given two vertices)
- Jarník's (Prim's) algorithm (find the minimal spanning tree)

d-ary heap

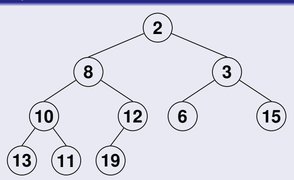
d-ary heap

- Every node has at most d children
- Every level except the last is completely filled
- The last level is filled from the left

Binary heap

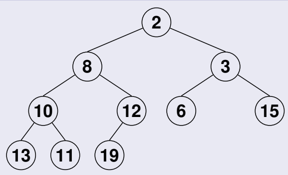
Binary heap is a 2-ary heap

Example of a binary heap



d-ary heap: Representation

Binary heap stored in a tree



Binary heap stored in an array

A node at index i has its parent at $\lfloor (i-1)/2 \rfloor$ and children at $2i + 1$ and $2i + 2$.

2	8	3	10	12	6	15	13	11	19

Parent

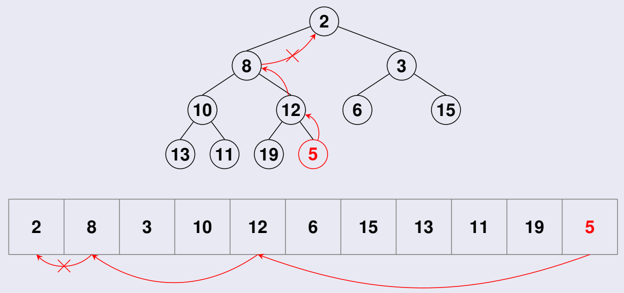
Children

d-ary heap: Height of the tree

- Nodes in an i-th level: d^i
- Maximal number of nodes in the d-ary heap of height h: $\sum_{i=0}^{h} d^i = \frac{d^{h+1}-1}{d-1}$
- Minimal number of nodes in the d-ary heap of height h: $\frac{d^h-1}{d-1} + 1$
- The number of nodes satisfies: $\frac{d^h-1}{d-1} + 1 \leq n \leq \frac{d^{h+1}-1}{d-1}$
  $h < \log_d(1+(d-1)n) \leq h+1$
- The height of the d-ary heap is: $h = \lceil \log_d(1+(d-1)n) \rceil - 1 = \lfloor \log_d (d-1)n \rfloor = \Theta(\log_d n)$
- Specially, the height of the binary heap is: $h = \lfloor \log_2 n \rfloor$

d-ary heap: Insert and decrease key

Example: Insert 5



d-ary heap: Insert and decrease key

Insert: Algorithm

Input: A new element with a key x
$v \leftarrow$ the first empty block in the array
Store the new element to the block v
while *v is not the root and the parent p of v has a key greater than x* **do**
  Swap elements v and p
  $v \leftarrow p$

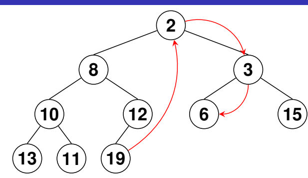
Decrease key (of a given node)

Decrease the key and swap the element with parents when necessary (likewise in the operation insert).

Complexity

$\mathcal{O}(\log_d n)$

d-ary heap: Delete min



Algorithm

Move the last element to the root v
while *Some children of v has smaller key than v* **do**
  $u \leftarrow$ the child of v with the smallest key
  Swap elements u and v
  $v \leftarrow u$

Complexity

If d is a fix parameter: $\mathcal{O}(\log n)$
If d is a part of the input: $\mathcal{O}(d \log_d n)$

d-ary heap: Building

Goal

Initialize a heap from a given array of elements

Algorithm

for *$r \leftarrow$ the last block* **to** *the first block* **do**
```
  # Heapify likewise in the operation delete
```
  $v \leftarrow r$
  while *Some children of v has smaller key than v* **do**
    $u \leftarrow$ the child of v with the smallest key
    Swap elements u and v
    $v \leftarrow u$

Correctness

After processing node r, its subtree satisfies the heap property.

d-ary heap: Building

Lemma

$$\sum_{h=0}^{\infty} \frac{h}{d^h} = \frac{d}{(d-1)^2}$$

Complexity

- Heapify a subtree with height h: $\mathcal{O}(h)$
- The number of nodes at height h is at most $\lceil d^{\log_d n - h - 1} \rceil = \lceil \frac{n}{d^{h+1}} \rceil$
- Total time complexity is

$$\sum_{h=0}^{\lceil \log_d n \rceil} \frac{n}{d^{h+1}} \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{d^h} \right) = \mathcal{O}(n)$$

Binomial tree

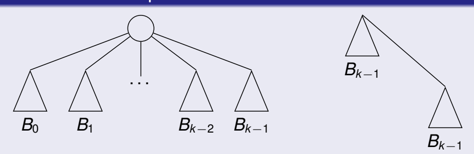
Definition

- A binomial tree B_0 of order 0 is a single node.
- A binomial tree B_k of order k has a root node whose children are roots of binomial trees of orders $0, 1, \ldots, k-1$.

Alternative definition

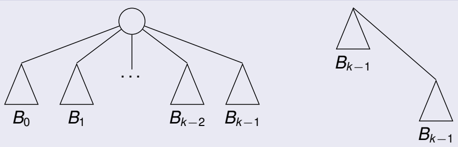
A binomial tree of order k is constructed from two binomial trees of order $k-1$ by attaching one of them as the rightmost child of the root of the other tree.

Recursions for binomial heaps

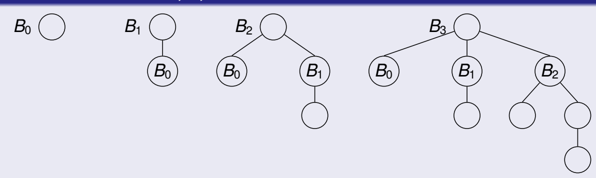


Binomial tree: Example

Recursions for binomial heaps

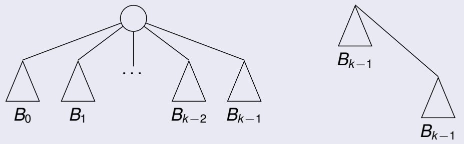


Binomial trees of order 0, 1, 2 and 3



Binomial tree: Properties

Recursions for binomial heaps



Observations

A binomial tree B_k has

- 2^k nodes,
- height k,
- k children in the root,
- maximal degree k,
- $\binom{k}{d}$ nodes at depth d.

Set of binomial trees

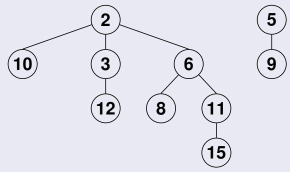
Observations

For every n there exists a set of binomial trees of pairwise different order such that the total number of nodes is n.

Relation between a binary number and a set of binomial trees

Binary number n	$=$	1	0	0	1	1	0	1	0
		↓			↓	↓		↓	
Binomial heap contains:		B_7			B_4	B_3		B_1	

Example of a set of binomial trees on 1010_2 nodes



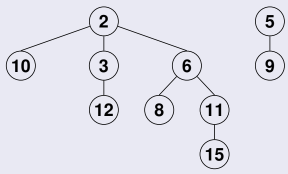
Binomial heap

Binomial heap

A binomial heap is a set of binomial trees that satisfies

- Each binomial tree obeys the minimum-heap property: the key of a node is greater than or equal to the key of its parent.
- There is at most one binomial tree for each order.

Example



Binomial heap: Height and size

Observation

Binomial heap contains at most $\log_2(n+1)$ trees and each tree has height at most $\log_2 n$.

Relation between a binary number and a set of binomial trees

Binary number n	$=$	1	0	0	1	1	0	1	0
		↓			↓	↓		↓	
Binomial heap contains:		B_7			B_4	B_3		B_1	

Binomial heap: Representation

A node in a binomial tree contains

- an element (key and value),
- a pointer to its parent,
- a pointer to its most-left child,
- a pointer to its right sibling and
- the number of children.

Binomial trees in a binomial heap

Binomial trees are stored in a linked list.

Remarks

- The child and the sibling pointers form a linked list of all children.
- Sibling pointers of all roots are used for the linked list of all trees in a binomial heap.

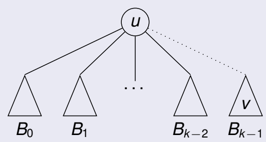
Binomial heap: Operations Decrease-key and Simple join

Decrease-key

Decrease the key and swap its element with parents when necessary (likewise in a binary heap).

Simple join

Two binomial trees B_{k-1} of order $k-1$ can be joined into B_k in time $\mathcal{O}(1)$.



The following values need to be set:

- the child pointer in the node u,
- the parent and the sibling pointers in the node v and
- the number of children in the node u.

Binomial heap: Operations Join and Insert

Join

It works as an analogy to binary addition. We start from the lowest orders, and whenever we encounter two trees of the same order, we join them.

Example

Binomial tree	B_6	B_5	B_4	B_3	B_2	B_1	B_0
First binomial heap	0	1	1	0	1	1	0
Second binomail heap	0	1	1	0	1	0	0
Joined binomial heap	1	1	0	1	0	1	0

Complexity of operation Insert

Complexity is $\mathcal{O}(\log n)$ where n is the total number of nodes.

Insert

Insert is implemented as join with a new tree of order zero.

- The worst-case complexity is $\mathcal{O}(\log n)$.
- The amortized complexity is $\mathcal{O}(1)$ — likewise increasing a binary counter.

Binomial heap: Operations Find-min and Delete-min

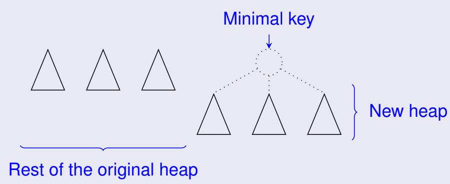
Find-min

$\mathcal{O}(1)$ if a pointer to the tree with the smallest key is stored, otherwise $\mathcal{O}(\log n)$.

Delete-min

Split the tree with the smallest key into a new heap by deleting its root and join the new heap with the rest of the original heap. The complexity is $\mathcal{O}(\log n)$.

Example



Lazy binomial heap

Difference
Lazy binomial heap is a set of binomial trees, i.e. different orders of binomial trees in a lazy binomial heap is not required.

Join and insert
Just concatenate lists of binomial trees, so the worst-case complexity is $\mathcal{O}(1)$.

Delete min
- Delete the minimal node
- Append its children to the list of heaps
- Reconstruct to the proper binomial heap

Lazy binomial heap: Reconstruction to the proper binomial heap

Idea
- While the lazy binomial heap contains two heaps of the same order, join them.
- Use an array indexed by the order to find heaps of the same order.

Algorithm
```
Initialize an array of pointers of size ⌈log_2 n⌉
for each heap h in the lazy binomial heap do
    o ← order of h
    while array[o] is not NIL do
        h ← the join of h and array[o]
        array[o] ← NIL
        o ← o + 1
    array[o] ← h
Create a binomial heap from the array
```


Lazy binomial heap: Reconstruction to the proper binomial heap

Worst-case complexity
- The original number of trees is at most n.
- Every iteration of the while-loop decreases the number of trees by one.
- The while-loop is iterated at most n-times.
- Therefore, the worst-case complexity is $\mathcal{O}(n)$.

Amortized complexity
- Consider the potential function $\Phi =$ *the number of trees*.
- The insert takes $\mathcal{O}(1)$-time and increases the potential by 1, so its amortized time is $\mathcal{O}(1)$.
- One iteration of the while-loop takes $\mathcal{O}(1)$-time and decreases the potential by 1, so its amortized time is zero.
- The remaining steps takes $\mathcal{O}(\log n)$-time.
- Therefore, the amortized time is $\mathcal{O}(\log n)$.

Heap: Overview

Complexity table

	Binary	Binomial		Lazy binomial	
	worst	worst	amort	worst	amort
Insert	$\log n$	$\log n$	1	1	1
Decrease-key	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
Delete-min	$\log n$	$\log n$	$\log n$	n	$\log n$

Question
Can we develop a heap with faster delete-min than $\mathcal{O}(\log n)$ and insert in time $\mathcal{O}(1)$?

Next goal
We need faster operation Decrease-key.

How?
If we relax the condition on trees in a binomial heap to be isomorphic to binomial trees, is there a faster method to decrease the key of a given node?

Fibonacci heap

Description
- Fibonacci heap is a set of trees.
- Each tree obeys the minimum-heap property.
- The structure of a Fibonacci heap follows from its operations.

Representation
Node of a Fibonacci heap contains
- an element (key and value),
- a pointer to its parent,
- a pointer to its most-left child,
- a pointer to its left and right sibling,
- the number of children and
- a flag which is set when the node losses a child.

Fibonacci heap is a linked list of trees.

Fibonacci heap: Operations

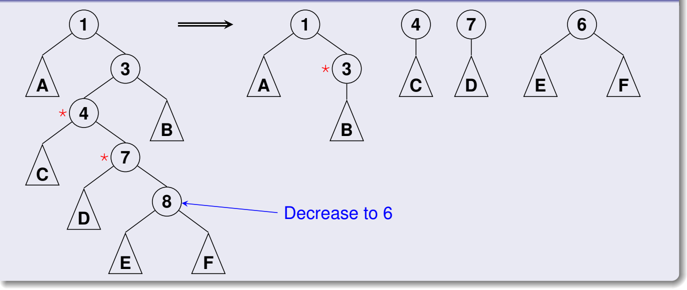
Join
Concatenate lists of trees. Complexity $\mathcal{O}(1)$.

Insert
Append a single node tree to the list of trees. Complexity $\mathcal{O}(1)$.

Flag
- Every node except roots can lose at most one child.
- When a node u losses a child, the flag in u is set.
- When u losses a second child, u is severed from its parent and whole subtree is inserted to the list of trees.
- This separation may lead to a *cascading cut*.
- Every root is unmarked.

Fibonacci heap: Decrease-key

Example



Fibonacci heap: Decrease-key

Algorithm
```
Input: A node u and new key k
Decrease key of the node u
if u is a root or the parent of u has key at most k then
    return # The minimal heap property is satisfied
p ← the parent of u
Unmark the flag in u
Remove u from its parent p and append u to the list of heaps
while p is not a root and the flag in p is set do
    u ← p
    p ← the parent of u
    Unmark the flag in u
    Remove u from its parent p and append u to the list of heaps
if p is not a root then
    Set the flag in p
```

Fibonacci heap: Delete-min

Idea from lazy binomial heaps

- Binomial heap joins two binomial trees of the same order.
- In Fibonacci heap, the order of a tree is the number of children of its root.

Algorithm

```
Input: A node u to be deleted
Delete the node u and append its children to the list of trees
# Reconstruction likewise in lazy binomial heap
Initialize an array of pointers of a sufficient size
for each tree t in the Fibonacci heap do
    c ← the number of children of the root of t
    while array[c] is not NIL do
        t ← the join of t and array[c]
        array[c] ← NIL
        c ← c + 1
    array[c] ← t
Create a Fibonacci heap from the array
```


Fibonacci heap: Fibonacci numbers

Definition

- $F_0 = 0$
- $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$ for $k \geq 2$

Properties

- $\sum_{i=1}^{k} F_i = F_{k+2} - 1$
- $F_k = \frac{(1+\sqrt{5})^k - (1-\sqrt{5})^k}{2^k \sqrt{5}}$
- $F_k \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$

Proof

A straightforward application of the mathematical induction.

Fibonacci heap: Structure

Invariant

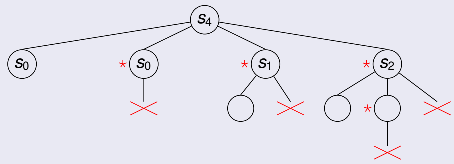
For every node u and its i-th child v holds that v has at least

- $i - 2$ children if v is marked and
- $i - 1$ children if v is not marked.

Size of a subtree

Let s_k be the minimal number of nodes in a subtree of a node with k children.
Observe that $s_k \geq s_{k-2} + s_{k-3} + s_{k-4} + \cdots + s_2 + s_1 + s_0 + s_0 + 1$.

Example



Fibonacci heap: Structure

Size of a subtree

Let s_k be the minimal number of nodes in a subtree of a node with k children.
Observe that $s_k \geq s_{k-2} + s_{k-3} + s_{k-4} + \cdots + s_2 + s_1 + s_0 + s_0 + 1$

Fibonacci numbers

- $F_0 = 0$ and $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$
- $\sum_{i=1}^{k} F_i = F_{k+2} - 1$
- $F_k = \frac{(1+\sqrt{5})^k - (1-\sqrt{5})^k}{2^k \sqrt{5}}$
- $F_k \geq \left(\frac{1+\sqrt{5}}{2}\right)^k$
- $s_k \geq F_{k+2}$

Corollary

A tree of order k has at least $s_k \geq F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^{k+2}$ nodes. Therefore,

- root of a tree on m nodes has $\mathcal{O}(\log m)$ children and
- Fibonacci heap has $\mathcal{O}(\log n)$ trees after operation Delete-min.

Fibonacci heap: Complexity

Worst-case complexity

- Operation Insert: $\mathcal{O}(1)$
- Operation Decrease-key: $\mathcal{O}(\log n)$
- Operation Delete-min: $\mathcal{O}(n)$

Amortized complexity: Potential

$\Phi = t + 2m$ where t is the number of trees and m is the number of marked nodes

Amortized complexity: Insert

- cost: $\mathcal{O}(1)$
- $\Delta\Phi = 1$
- Amortized complexity: $\mathcal{O}(1)$

Fibonacci heap: Amortized complexity of Decrease-key

Single iteration of the while-loop (unmark and cut)

- Cost: $\mathcal{O}(1)$
- $\Delta\Phi = 1 - 2 = -1$
- Amortized complexity: Zero

Remaining parts

- Cost: $\mathcal{O}(1)$
- $\Delta\Phi \leq 1$
- Amortized complexity: $\mathcal{O}(1)$

Total amortized complexity

$\mathcal{O}(1)$

Fibonacci heap: Amortized complexity of Delete-min

Delete root and append its children

- Cost: $\mathcal{O}(\log n)$
- $\Delta\Phi \leq \mathcal{O}(\log n)$
- Amortized complexity: $\mathcal{O}(\log n)$

Single iteration of the while-loop (join)

- Cost: $\mathcal{O}(1)$
- $\Delta\Phi = -1$
- Amortized complexity: Zero

Remaining parts

- Cost: $\mathcal{O}(\log n)$
- $\Delta\Phi = 0$
- Amortized complexity: $\mathcal{O}(\log n)$

Total amortized complexity

$\mathcal{O}(\log n)$

Appending all children of the root can be done in $\mathcal{O}(1)$ by a simple concatenating of linked lists. However, some of these children can be marked, so unmarking takes $\mathcal{O}(\log n)$-time as required by our definition. In a practical implementation, it is not important when flags of roots are unmarked.

Heap: Overview

Complexity table

	Binary	Binomial		Lazy binomial		Fibonacci	
	worst	worst	amort	worst	amort	worst	amort
Insert	$\log n$	$\log n$	1	1	1	1	1
Decrease-key	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	1
Delete-min	$\log n$	$\log n$	$\log n$	n	$\log n$	n	$\log n$

Heaps: Dijkstra's algorithm

Problem

Given a graph $G = (V, E)$ with non-negative weight on edges ω and a starting vertex s, find the shortest paths from s to all vertices.

Algorithm

```
Create an empty priority queue Q for vertices of G
for v ← V do
    distance[v] ← 0 if v = s else ∞
    Insert v with the key distance[v] into Q
while Q is non-empty do
    Extract the vertex u with the smallest key (distance) from Q
    for v ← neighbour of u do
        if distance[v] > distance[u] + ω(u, v) then
            distance[v] ← distance[u] + ω(u, v)
            Decrease the key of v in Q
```


Heaps: Dijkstra's algorithm

Number of operations

Dijkstra's algorithm may call

- operation Insert for every vertex,
- operation Delete-min for every vertex and
- operation Decrease-key for every edge.

We assume that $m \geq n$ where $n = |V|$ and $m = |E|$.

Complexity table

	Array	Binary	Binomial	Fibonacci	k-ary
Insert	1	$\log n$	1	1	$\log_k n$
Delete-min	n	$\log n$	$\log n$	$\log n$	$k \log_k n$
Decrease-key	1	$\log n$	$\log n$	1	$\log_k n$
Dijkstra's	n^2	$m \log n$	$m \log n$	$m + n \log n$	$m \log_{m/n} n$

Linear-time complexity

- When $m/n = \Theta(m)$ using an array.
- When $m/n = \Omega(n^\epsilon)$ using a k-ary heap.
- When $m/n = \Omega(\log n)$ using a Fibonacci heap.

- The complexity for k-ary heap is $\mathcal{O}(nk \log_k n + m \log_k n)$.
  Both terms are equal for $k = m/n$.
  The term $\log_{m/n} n$ is $\mathcal{O}(1)$ if $m \geq n^{1+\epsilon}$ for some $\epsilon > 0$.
- Using a Fibonacci heap is inefficient in practice.
- Monotonic heaps (e.g. Thorup heap) have Delete-min in time $\mathcal{O}(\log \log n)$, so Dijkstra's algorithm runs in $\mathcal{O}(m + \log \log n)$.
- More details are presented in the course "Graph Algorithms" by Martin Mareš.

Outline

Techniques for memory hierarchy

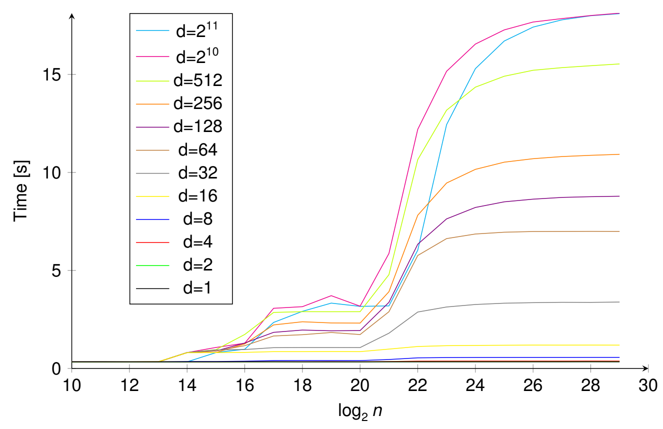
Example of sizes and speeds of different types of memory

	size	speed
L1 cache	32 KB	223 GB/s
L2 cache	256 KB	96 GB/s
L3 cache	8 MB	62 GB/s
RAM	32 GB	23 GB/s
HDD 1	112 GB	56 MB/s
HDD 2	2 TB	14 MB/s
Internet	∞	10 MB/s

A trivial program

```
for (i=0; i+d<n; i+=d) do
    A[i] = i+d
A[i]=0
for (j=0; j< 2^28; j++) do
    i=A[i]
```


Memory models: A trivial program



Memory models

1. For simplicity, consider only two types of memory called a disk and a cache.
2. Memory is split into pages of size B. ①
3. The size of the cache is M, so it can store $P = \frac{M}{B}$ pages.
4. CPU can access data only in cache.
5. The number of page transfers between disk and cache in counted. ②
6. For simplicity, the size of one element is unitary. ③

External memory model

Algorithms explicitly issues read and write requests to the disks, and explicitly manages the cache.

Cache-oblivious model

Design external-memory algorithms without knowing M and B. Hence,

- a cache oblivious algorithm works well between any two adjacent levels of the memory hierarchy,
- no parameter tuning is necessary which makes programs portable,
- algorithms in the cache-oblivious model cannot explicitly manage the cache.

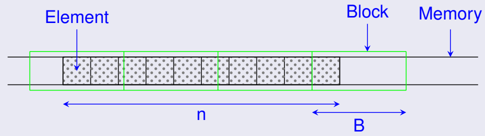
Cache is assumed to be fully associative.

1. Also called a block or a line.
2. For simplicity, we consider only loading pages from disk to cache, which is also called page faults.
3. Therefore, B and M are the maximal number of elements in a page and cache, respectively.

Cache-oblivious analysis: Scanning

Scanning

Traverse all elements in an array, e.g. to compute sum or maximum.



- The optimal number of page transfers is $\lceil n/B \rceil$.
- The number of page transfers is at most $\lceil n/B \rceil + 1$.

Array reversal

Assuming $P \geq 2$, the number of page transfers is the same. ①

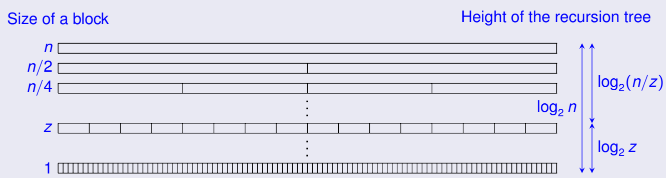
1. We also assume that CPU has a constant number of registers that stores loop iterators, $\mathcal{O}(1)$ elements, etc.

Cache-oblivious analysis: Mergesort

Case $n \leq M/2$

Whole array fits into cache, so $2n/B + \mathcal{O}(1)$ page are transfered. ①

Schema



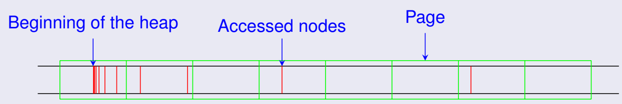
Case $n > M/2$

1. Let z be the maximal block in the recursion that can be sorted in cache.
2. Observe: $z \leq \frac{M}{2} < 2z$
3. Merging one level requires $2\frac{n}{B} + 2\frac{n}{z} + \mathcal{O}(1) = \mathcal{O}\left(\frac{n}{B}\right)$ page transfers. ②
4. Hence, the number of page transfers is $\mathcal{O}\left(\frac{n}{B}\right)\left(1 + \log_2 \frac{n}{z}\right) = \mathcal{O}\left(\frac{n}{B} \log \frac{n}{M}\right)$. ③

1. Half cache is for two input arrays and the other half is for the merged array.
2. Merging all blocks in level i into blocks in level $i-1$ requires reading whole array and writing the merged array. Furthermore, misalignments may cause that some pages contain elements from two blocks, so they are accessed twice.
3. Funnelsort requires $\mathcal{O}\left(\frac{n}{B} \log_P \frac{n}{B}\right)$ page transfers.

Cache-oblivious analysis: Binary heap and search

Binary heap: A walk from the root to a leaf



1. The path has $\Theta(\log n)$ nodes.
2. First $\Theta(\log B)$ nodes on the path are stored in at most two pages. ①
3. Remaining nodes are stored in pair-wise different pages.
4. $\Theta(\log n - \log B)$ pages are transfered. ②

Binary search

- $\Theta(\log n)$ elements are compared with a given key.
- Last $\Theta(\log B)$ nodes are stored in at most two pages.
- Remaining nodes are stored in pair-wise different pages.
- $\Theta(\log n - \log B)$ pages are transfered.

1. One page stores B nodes, so the one page stores a tree of height $\log_2(B) + \mathcal{O}(1)$, if the root is well aligned.
2. More precisely: $\Theta(\max\{1, \log n - \log B\})$

Cache-oblivious analysis: Cache-aware search

Search in a balanced binary search tree

Height of a tree is $\Theta(\log n)$, so $\Theta(\log n)$ pages are transfered. ①

Cache-aware algorithm

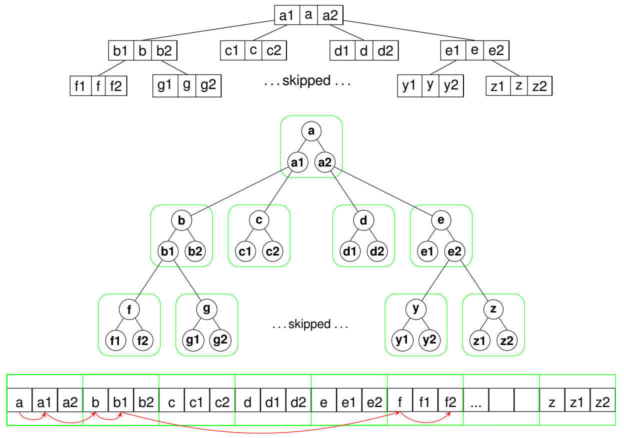
Cache-aware algorithms use exact values of sizes of a page and cache.

Search in an (a,b)-tree and cache-aware binary tree

- Choose a and b so that the size of one node of an (a,b)-tree is exactly B.
- Height of the (a,b)-tree is at most $\log_a n + \mathcal{O}(1)$.
- Search from the root to a leaf requires only $\Theta(\log_B n)$ page transfers. ②
- Replace every node of the (a,b)-tree by a binary subtree stored in one memory page. ③
- A search in this binary tree requires also $\Theta(\log_B n)$ page transfers. ④
- However, we would prefer to be independent on B.

1. When nodes are allocated independently, nodes on a path from the root to a leaf can be stored in different pages.
2. The height would be between $\log_b n$ and $1 + \log_a n$ and these bounds would be equal to $\Theta(\log_B n)$.
3. Assuming whole subtree also fits into a single memory page.
4. This is also the best possible (the proof requires Information theory).

Cache-oblivious analysis: Cache-aware representation



Path from the root to the leaf $f2$

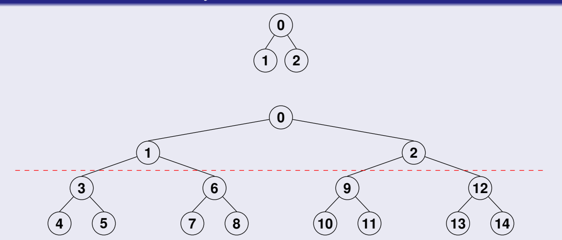
Cache-oblivious analysis: The van Emde Boas layout

Recursive description

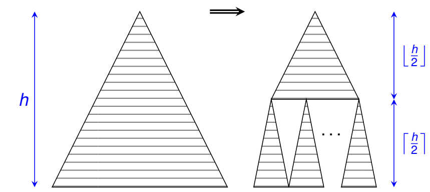
- Van Emde Boas layout of order 0 is a single node.
- The layout of order k has one "top" copy of the layout of order $k-1$ and every leaf of the "top" copy has attached roots of two "bottom" copies of the layout of order $k-1$ as its children.

All nodes of the tree are stored in an array so that the "top" copy is the first followed by all "bottom" copies.

The order of nodes in the array



Cache-oblivious analysis: The van Emde Boas layout



Number of page transfers

- Let $h = \log_2 n$ be the height of the tree.
- Let z be the maximal height of a subtree in the recursion that fits into one page.
- Observe: $z \le \log_2 B \le 2z$.
- The number of subtrees of height z on the path from the root to a leaf is $\frac{h}{z} \le \frac{2\log_2 n}{\log_2 B} = 2\log_B n$
- Hence, the number of page transfers is $\mathcal{O}(\log_B n)$.

- What is the number of subtrees?
- What is the number of nodes in each subtree?
- Is there a simple formula to determine indices of the parent and children for a given index of an element in this array?
- Find algorithm which returns indices of the parent and children for a given index of an element. Is there a faster algorithm than $\mathcal{O}(\log\log n)$?
- Find algorithm which for a given node u write all nodes of the path from u to the root in time linear in the length of the path.

Cache-oblivious analysis: The van Emde Boas layout: Initialization

Initialize of an array A to form the van Emde Boas layout ①

```
Function Init(A, n, root_parent) ②
    L ← empty
    if n == 1 then
        A[0].parent ← root_parent
        A[0].children[0], A[0].children[1] ← NULL
    else
        k ← min_z such that 2^(2^z) > n ③
        s ← 2^(2^(k-1)) − 1 ④
        P ← Init(A, s, root_parent) ⑤
        C ← A + s ⑥
        i ← 0 ⑦
        while C < A + n do
            L.append(Init(C, min{s, A + n − C}, P + ⌊i/2⌋)) ⑧
            P[⌊i/2⌋].children[i mod 2] ← C
            C ← C+s ⑨
            i ← i+1
    return L
```


1. Every element of the array contains pointers to its parent and children.
2. n is the size of array to be initialized
Returns a list of all leaves
3. The minimal number of subdivision of the binary tree on n nodes to reach trivial subtrees
4. Number of nodes in every subtree after one subdivision
5. Initialize the top subtree. Leaves of the top subtree are roots of bottom subtrees.
6. The root of the first bottom subtree
7. Index of bottom subtrees
8. Initialize the i-th bottom subtree
9. Move to the next subtree

Cache-oblivious analysis: Matrix transposition: Simple approach

Page replacement strategies

- Optimal: The future is known, off-line
- LRU: Evicting the least recently used page
- FIFO: Evicting the oldest page

Simple algorithm for a transposing matrix A of size $k \times k$

```
for i ← 2 to k do
    for j ← i + 1 to k do
        Swap(A_ij, A_ji)
```

Assumptions

For simplicity, we assume that $B < k$ and $P < k$. ①

The number of page transfers by the simple algorithm

- Optimal page replacement: $\Omega\left((k-P)^2\right)$
- LRU or FIFO: $\Omega(k^2)$

1. One page stores at most one row of the matrix and cache cannot store all elements of one column at once.

Cache-oblivious analysis: Matrix transposition: Simple approach

Representation of a matrix 5×5 in memory and an example of memory pages

11	12	13	14	15	21	22	23	24	25	31	32	33	34	35	41	42	43	44	45	51	52	53	54	55

Optimal page replacement

1. Transposing the first row requires at least k transfers.
2. Then, at most P elements of the second column is cached.
3. Therefore, transposing the second row requires at least $k - P - 1$ transfers.
4. Transposing the i-th row requires at least $\max\{0, k - P - i\}$ transfers.
5. The total number of transfers is at least $\sum_{i=1}^{k-P} i = \Omega\left((k-P)^2\right)$.

LRU or FIFO page replacement

All the column values are evicted from the cache before they can be reused, so $\Omega(k^2)$ pages are transfered.

Cache-oblivious analysis: Matrix transposition: Optimal approach

Idea

Recursively split the matrix into sub-matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad A^T = \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix}$$

Number of page transfers

1. Tall cache assumption: $M \geq B^2$
2. Let z be the maximal size of a sub-matrix in the recursion that fit into cache.
3. Observe: $z \leq B \leq 2z$
4. There are $(k/z)^2$ sub-matrices of size z.
5. Transposition two such sub-matrices requires $\mathcal{O}(z)$ transfers.
6. The number of transfers is $\mathcal{O}(k^2/B)$.
7. This approach is optimal up-to a constant factor.

How this matrix transposition can be implemented without recursion nor stack?

Cache-oblivious analysis: Comparison of LRU and OPT strategies

Theorem (Sleator, Tarjan, 1985)

- Let $s_1, \dots, s_k$ be a sequence of pages accessed by an algorithm.
- Let n_{OPT} and n_{LRU} be the number of pages in cache for OPT and LRU, resp.
- Let F_{OPT} and F_{LRU} be the number of page faults during the algorithm.

Then, $F_{LRU} \leq \frac{n_{LRU}}{n_{LRU} - n_{OPT}} F_{OPT} + n_{OPT}$.

Corollary

If LRU can use twice as many cache pages as OPT, then LRU transports at most twice many pages than OPT does.

The asymptotic number of page faults for some algorithms

In most cache-oblivious algorithms, doubling/halving cache size has no impact on the asymptotic number of page faults, e.g.

- Scanning: $\mathcal{O}(n/B)$
- Mergesort: $\mathcal{O}(\frac{n}{B} \log \frac{n}{M})$
- Funnelsort: $\mathcal{O}(\frac{n}{B} \log_P \frac{n}{B})$
- The van Emde Boas layout: $\mathcal{O}(\log_B n)$

Cache-oblivious analysis: Comparison of LRU and OPT strategies

Theorem (Sleator, Tarjan, 1985)

1. Let $s_1, \dots, s_k$ be a sequence of pages accessed by an algorithm.
2. Let n_{OPT} and n_{LRU} be the number of pages in cache for OPT and LRU, resp.
3. Let F_{OPT} and F_{LRU} be the number of page faults during the algorithm.

Then, $F_{LRU} \leq \frac{n_{LRU}}{n_{LRU} - n_{OPT}} F_{OPT} + n_{OPT}$.

Proof

1. A subsequence of s, which LRU faults the same page twice, contains at least $n_{LRU} + 1$ different pages.
2. If LRU faults $f \leq n_{LRU}$ pages during a subsequence of s, then the subsequence accesses at least f different pages and OPT faults at least $f - n_{OPT}$ pages during the subsequence.
3. Split the sequence of s into subsequences such that LRU has exactly n_{LRU} page faults during each subsequence (except one).
4. OPT has at least $n_{LRU} - n_{OPT}$ faults during each subsequence (except one).
5. The additive term "$+n_{OPT}$" in the theorem is necessary for the exceptional subsequence in which LRU may have less than n_{LRU} page faults.

Cache-oblivious analysis: Other algorithms and data structures

- Funnelsort
- Long integer multiplication
- Matrix multiplication
- Fast Fourier transform
- Dynamic B-trees
- Priority queues
- kd-tree

Outline

Hash tables

Basic terms

- Universe $U = \{0, 1, \dots, u-1\}$ of all elements
- Represent a subset $S \subseteq U$ of size n
- Store S in an array of size m using a hash function $h : U \to M$ where $M = \{0, 1, \dots, m-1\}$
- Collision of two elements $x, y \in S$ means $h(x) = h(y)$
- Hash function h is perfect on S if h has no collision on S

Adversary subset

If $u \geq mn$, then for every hashing function h there exists $S \subseteq U$ of size n such that $|h(S)| = 1$.

Birthday paradox

When n balls are (uniformly and independently) thrown into $m \geq n$ bins, the probability that every bin has at most one ball is

$$\prod_{i=1}^{n-1} \frac{m-i}{m} \sim e^{-\frac{n^2}{2m}}.$$

Hash tables

Basic issues

- Find a good hash function
- Handle collisions

Simple hash function

$h(x) = x \mod m$

- \+ Fast to compute
- \+ $h^{-1}(j)$ have almost the same size for all $j \in M$
- \+ Works well only if the input is random
- \- The adversary subset is easily determined (DOS attack)

Cryptographic hash function, e.g. MD5, SHA-1

- \+ Hard to deliberately find a collision
- \- Slow and complex

Totally random hash function (assumed in analysis of hash tables)

Values $h(x)$ for $x \in S$ are assumed to be independent random variables with the uniform distribution on M.

Hash tables: Separate chaining

Description

Bucket j stores all elements $x \in S$ with $h(x) = j$ using some data structure, e.g.

- a linked lists
- a dynamic array
- a self-balancing tree

Implemetations

- std::unordered_map in C++
- Dictionary in C#
- HashMap in Java
- Dictionary in Python

Hash tables: Separate chaining: Example

Using illustrative hash function $h(x) = x \mod 11$

0, 22, 55
2
14, 80
5, 27
17
8, 30
21

1. 37% buckets are empty for $\alpha = 1$
2. Successful search: The total number of comparison to find all elements in the table is computed by summing over all buckets the number of comparisons to find all elements in a bucket, that is $\sum_j \sum_{k=1}^{A_j} k = \sum_j \frac{A_j(A_j+1)}{2}$. Hence, the expected number of comparisons is
$\frac{1}{n}\sum_j \frac{A_j(A_j+1)}{2} = \frac{1}{2} + \frac{m}{2n}\frac{\sum_j A_j^2}{m} = \frac{1}{2} + \frac{1}{2\alpha}E[A_j^2] = 1 + \frac{\alpha}{2} - \frac{1}{2m}$.
Unsuccessful search: Assuming that uniformly distributed random bucket is search, the number of comparisons is $E[A_j] = \alpha$.

Hash tables: Separate chaining: Analysis

Definition

- $\alpha = \frac{n}{m}$ is the load factor
- I_{ij} is a random variable indicating whether i-th element belongs into j-th bucket
- $A_j = \sum_{i \in S} I_{ij}$ is the number of elements in j-th bucket

Basic observations

1. $E[A_j] = \alpha$
2. $E[A_j^2] = \alpha(1 + \alpha - 1/m)$
3. $Var(A_j) = \alpha(1 - 1/m)$
4. $\lim_{n\to\infty} P[A_j = 0] = e^{-\alpha}$ ①

Number of comparisons in operation Find

The expected number of key comparisons is α for the unsuccessful search and $1 + \frac{\alpha}{2} - \frac{1}{2m}$ for the successful search. Hence, the average complexity of Find is $\mathcal{O}(1+\alpha)$. ②

Hash tables: Separate chaining: Analysis

Definition

An event E_n whose probability depends on a number n occurs with high probability if there exists a constant $c > 0$ and an integer n_0 such that $P[E_n] \geq 1 - \frac{1}{n^c}$ for every $n \geq n_0$.

Chernoff Bound

Suppose $X_1, \dots, X_n$ are independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $\mu = E[X]$ denote the sum's expected value. Then for any $c > 1$ holds

$$P[X > c\mu] < \frac{e^{(c-1)\mu}}{c^{c\mu}}.$$

Upper bound on the longest chain

Assuming $\alpha = \Theta(1)$, every bucket has $\mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ elements with high probability.

Expected length of the longest chain (without a proof)

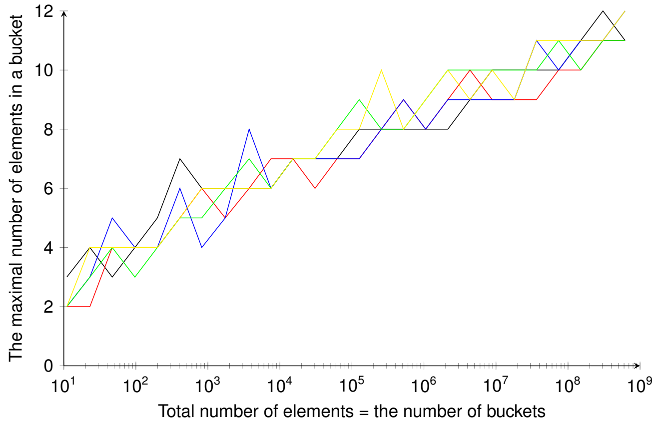
Assuming $\alpha = \Theta(1)$, the expected length of the longest chain is $\Theta(\frac{\log n}{\log\log n})$ elements.

Let $\epsilon > 0$ and $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$. We have to estimate $P[\max_j A_j > c\mu]$. Observe that $P[\max_j A_j > c\mu] \leq \sum_j P[A_j > c\mu] = mP[A_1 > c\mu]$. We apply Chernoff bound on variables I_{1i} to obtain

$$\begin{aligned}
P[A_1 > c\mu] &< e^{-\mu} e^{c\mu - c\mu \log c} \\
&= e^{-\mu} e^{(1+\epsilon)\frac{\log n}{\log\log n} - (1+\epsilon)\frac{\log n}{\log\log n}\log\left(\frac{(1+\epsilon)\log n}{\mu \log\log n}\right)} \\
&= e^{-\mu} e^{(1+\epsilon)\frac{\log n}{\log\log n} - (1+\epsilon)\log n + (1+\epsilon)\frac{\log n}{\log\log n}\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)} \\
&= \frac{1}{n^{1+\frac{\epsilon}{2}}} e^{-\mu} n^{-\frac{\epsilon}{2} + (1+\epsilon)\frac{1}{\log\log n} + (1+\epsilon)\frac{\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)}{\log\log n}} \\
&< \frac{1}{n^{1+\frac{\epsilon}{2}}}
\end{aligned}$$

Indeed, both $\frac{1}{\log\log n}$ and $\frac{\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)}{\log\log n}$ converge to zero, so for sufficiently large n the power of n is negative. Hence, $P[\max_j A_j \leq (1+\epsilon)\frac{\log n}{\log\log n}] > 1 - \frac{\alpha}{n^{\frac{\epsilon}{2}}}$.

Hash tables: Separate chaining: Example



Hash tables: Separate chaining: Analysis

The worst case search time for one element

The expected time for operations Find in the worst case is $\mathcal{O}\left(\frac{\log n}{\log\log n}\right)$.

Goal

Amortized complexity of searching is $\mathcal{O}(1)$ with high probability.

Probability of k comparisons for searching one element

$\lim_{n\to\infty} P[A_j = k] = \frac{\alpha^k}{k! e^\alpha}$ 1

Lemma: Number of elements in $\Theta(\log n)$ buckets

Assuming $\alpha = \Theta(1)$ and given $d \log n$ buckets T where $d > 0$, the number of elements in T is at most $e\alpha d \log n$ with high probability. 2

Amortized complexity for searching $\Omega(\log n)$ elements (Pătraşcu [7])

Assuming $\alpha = \Theta(1)$ and a cache of size $\Theta(\log n)$, the amortized complexity for searching $\Omega(\log n)$ elements is $\mathcal{O}(1)$ with high probability. 3 4

1. $A_j = k$ if k elements from S falls into the bucket j and others $n-k$ elements falls into other buckets. Therefore,
$$P[A_j = k] = \binom{n}{k}\frac{1}{m^k}\left(1-\frac{1}{m}\right)^{n-k} \sim \frac{n^k}{k! m^k}\left(1-\frac{\alpha}{n}\right)^n \to \frac{\alpha^k}{k! e^\alpha}.$$
2. Let the indicator variable X_i is 1 if $h(i) \in T$ for $i \in S$. The number of elements in T is $X = \sum_i X_i$ with $\mu = E[X] = E[\sum_{j\in T} A_j] = |T| E[A_j] = \alpha d \log n$. Using Chernoff bound we obtain
$$P[X > c\mu] < \exp\{d\alpha \log n(c - 1 - c\log c)\} = n^{d\alpha(c-1-c\log c)} = n^{-d\alpha}$$
for $c = e$.
3. A sequence of $\Omega(\log n)$ operations Find can be split into subsequences of length $\log n$. Furthermore, we use a cache for last $\log n$ elements to avoid repetitive searching of elements in the same bucket.
4. If $\log n$ searched elements are chosen randomly, they belong to pair-wise different buckets with high probability (see the birthday paradox).

Hash tables: Separate chaining: Multiple-choice hashing

2-choice hashing

Element x can be stored in buckets $h_1(x)$ or $h_2(x)$ and Insert chooses the one with smaller number of elements where h_1 and h_2 are two hash functions.

2-choice hashing: Longest chain (without a proof)

The expected length of the longest chain is $\mathcal{O}(\log\log n)$.

d-choice hashing

Element x can be stored in buckets $h_1(x), \dots, h_d(x)$ and Insert chooses the one with smallest number of elements where $h_1, \dots, h_d$ are d hash functions.

d-choice hashing: Longest chain (without a proof)

The expected length of the longest chain is $\frac{\log\log n}{\log d} + \mathcal{O}(1)$.

Hash tables: Linear probing

Memory consumption for separate chaining

Separate chaining uses memory for n element and

- $m + n$ pointers if buckets are implemented using linked list, or
- m pointers and m integers if buckets use dynamic arrays.

Goal

Store elements directly in the table.

Linear probing

Insert a new element x into the empty bucket $h(x) + i \mod m$ with minimal $i \geq 0$ assuming $n \leq m$.

Operation Find

Iterate until the given key or empty bucket is fount.

Operation Delete

Flag the bucket of deleted element to ensure that the operation Find continues searching.

Hash tables: Linear probing: Analysis

Complexity of Insert and unsuccessful Find

For every $\alpha < 1$, the expected number of key comparisons in operations Insert and unsuccessful Find is $\mathcal{O}(1)$.

Chernoff Bound

Suppose $X_1, \dots, X_n$ are independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $\mu = E[X]$ denote the sum's expected value. Then for any $c > 1$ holds
$$P[X > c\mu] < \frac{e^{(c-1)\mu}}{c^{c\mu}}.$$

Better estimates (Knuth [4]) (without a proof)

The expected number of key comparisons is at most $\frac{1}{2}(1 + \frac{1}{1-\alpha})$ in a successful search $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ in a unsuccessful search and insert.

- Let $0 < \alpha < 1$ and $1 < c < \frac{1}{\alpha}$ and $q = \left(\frac{e^{c-1}}{c^c}\right)^\alpha$. Observe $0 < q < 1$.
- First, we estimate the probability p_t that t elements of S are hashed into T for given subset of buckets T of size t, that is $p_t = P[|h(S) \cap T| = k]$. In order to apply Chernoff bound, let X_i be the indicator variable that $h(i) \in T$ for all $i \in S$. Then, $\mu = t\alpha$. Hence, $p_t = P[X = t] \leq P[X > c\mu] < q^t$.
- Next, we estimate the probability p'_k that we need k probings to find an empty space. The inserted or searched element is hashed into a bucket b. Let k and s are numbers such that buckets $b - s - 1$ and $b + k$ are empty buckets and all buckets between $b - s$ and $b + k - 1$ are occupied. Hence, $p'_k \leq \sum_{s=0}^\infty p_{s+k} < q^k \sum_{s=0}^\infty q^s = \frac{q^k}{1-q}$.
- Finally, we estimate the expected number of probings which is at most $\sum_{k=0}^m kp'_k < \frac{1}{1-q}\sum_{k=0}^\infty kq^k = \frac{2-q}{(1-q)^3}$.

Hash tables: Other methods

Quadratic probing

Insert a new element x into the empty bucket $h(x) + ai + bi^2 \mod m$ with minimal $i \geq 0$ where a, b are fix constants.

Double hashing

Insert a new element x into the empty bucket $h_1(x) + ih_2(x) \mod m$ with minimal $i \geq 0$ where h_1 and h_2 are two hash functions.

Brent's variation for operation Insert

If the bucket

- $b = h_1(x) + ih_2(x) \mod m$ is occupied by an element y and
- $b + h_2(x) \mod m$ is also occupied but
- $c = b + h_2(y) \mod m$ is empty,

then move element y to c and insert x to b. This reduces the average search time.

Hash tables: Cuckoo hashing

Origin

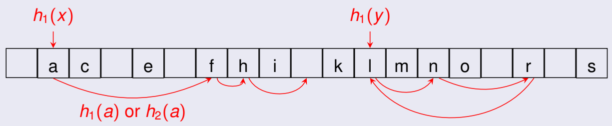
Rasmus Pagh and Flemming Friche Rodler [6]

Description

Given two hash functions h_1 and h_2, a key x can be stored in $h_1(x)$ or $h_2(x)$. Therefore, operations Find and Delete are trivial.

Insert: Example

- Successful insert of element x into $h_1(x)$ after three reallocations.
- Impossible insert of element y into $h_1(y)$.



Hash tables: Cuckoo hashing: Algorithm for Insert

Insert an element x into a hash table T

```
pos ← h1(x)
for n times do
    if T[pos] is empty then
        T[pos] ← x
        return
    swap(x, T[pos])
    if pos == h1(x) then
        pos ← h2(x)
    else
        pos ← h1(x)
rehash()
insert(x)
```

Rehashing

- Choose new hash functions h_1 and h_2
- Increase the size of the table if necessary
- Insert all elements to the new table

Hash tables: Cuckoo hashing: Analysis

Undirected cuckoo graph G

- Vertices are positions in the hash table.
- Edges are pairs $\{h_1(x), h_2(x)\}$ for all $x \in S$.

Properties of the cuckoo graph

- Operation Insert follows a path from $h_1(x)$ to an empty position.
- New element cannot be inserted into a cycle.
- When the path from $h_1(x)$ goes to a cycle, rehash is needed.

Lemma

Let $c > 1$ and $m \geq 2cn$. For given positions i and j, the probability that there exists a path from i to j and the shortest one has length k is at most $\frac{1}{mc^k}$.

Complexity of operation Insert without rehashing

Let $c > 1$ and $m \geq 2cn$. The expected length of the path is $\mathcal{O}(1)$.

Number of rehashes

Let $c > 2$ and $m \geq 2cn$. The expected number of rehashes is $\mathcal{O}(1)$.

Proof of the lemma by induction on k:

$k = 1$ For one element, the probability that it forms an edge ij is $\frac{2}{m^2}$. So, the probability that there is an edge ij is at most $\frac{2n}{m^2} \leq \frac{1}{mc}$.

$k > 1$ There exists a path between i and j of length k if there exists a path from i to u of length $k-1$ and an edge uj. For one position u, the i-u path exists with probability $\frac{1}{mc^{k-1}}$. The conditional probability that there exists the edge uj if there exists i-u path is at most $\frac{1}{mc}$ because some elements are used for the i-u path. By summing over all positions u, the probability that there exists i-j path is at most $m\frac{1}{mc^{k-1}}\frac{1}{mc} = \frac{1}{mc^k}$.

Insert without rehashing:

- Using the previous lemma for all length k and all end vertices j, the expected length of the path during operation Insert is $m\sum_{k=1}^{n} k\frac{1}{mc^k} \leq \sum_{k=1}^{\infty}\frac{k}{c^k} = \frac{c}{(c-1)^2}$.

Number of rehashes:

- Using the previous lemma for all length k and all vertices $i = j$, the probability that the graph contains a cycle is at most $m\sum_{k=1}^{\infty}\frac{1}{mc^k} = \frac{1}{c-1}$.
- The probability that inserting rehashes z times is at most $\frac{1}{(c-1)^z}$.
- The expected number of rehashes is at most $\sum_{z=0}^{\infty} z\frac{1}{(c-1)^z} = \frac{c-1}{(c-2)^2}$.

Hash tables: Cuckoo hashing: Analysis

Complexity operation Insert without rehashing

Let $c > 1$ and $m \geq 2cn$. The expected length of the path is $\mathcal{O}(1)$.

Amortized complexity of rehashing

Let $c > 2$ and $m \geq 2cn$. The expected number of rehashes is $\mathcal{O}(1)$.
Therefore, operation Insert has the expected amortized complexity $\mathcal{O}(1)$.

The estimation in the proof is not optimal

- The probability that the cuckoo graph has a cycle is overestimated.
- Rehashing is not necessary if the cuckoo graph has a cycle.

In fact, the expected number of rehashes is $\mathcal{O}(1)$ even for $c > 1$.

Summary

Find and Delete: $\mathcal{O}(1)$ worst case complexity
Insert: $\mathcal{O}(1)$ expected amortized complexity for $\alpha < 0.5$

Hash tables: Hash functions

Basic terms

- Universe $U = \{0, 1, \dots, u-1\}$ of all elements
- Represent a subset $S \subseteq U$ of size n
- Store S in an array of size m using a hash function $h : U \to M$ where $M = \{0, 1, \dots, m-1\}$

Hashing random data

Every reasonable function $f : U \to S$ is sufficient for hashing random data, e.g. $f(x) = x \bmod m$.

Random hash function

$u \log_2 m$ bits are necessary to represent a random hash function.

Adversary subset

If $u \geq mn$, then for every hashing function h there exists $S \subseteq U$ of size n such that $|h(S)| = 1$.

Hash tables: Universal hashing

Universal hashing

A set $\mathcal{H}$ of hash functions is universal if randomly chosen $h \in \mathcal{H}$ satisfies $P[h(x_1) = h(x_2)] \leq \frac{1}{m}$ for every $x_1 \neq x_2$ elements of U.

2-universal hashing

A set $\mathcal{H}$ of hash functions is 2-universal if randomly chosen $h \in \mathcal{H}$ satisfies $P[h(x_1) = z_1 \text{ and } h(x_2) = z_2] \leq \frac{1}{m^2}$ for every $x_1 \neq x_2$ elements of U and $z_1, z_2 \in M$.

k-universal hashing (also call k-wise independent)

A set $\mathcal{H}$ of hash functions is k-universal if randomly chosen $h \in \mathcal{H}$ satisfies $P[h(x_i) = z_i \text{ for every } i = 1, \dots, k] \leq \frac{1}{m^k}$ for every pair-wise different elements $x_1, \dots, x_k \in U$ and $z_1, \dots, z_k \in M$.

Relations

- If a function is k-universal, then it is also $k-1$ universal. ①
- If a function is 2-universal, then it is also universal. ②
- 1-universal function may not be universal. ③

1. $P[h(x_i) = z_i \text{ for every } i = 1, \dots, k-1]$
$= P[h(x_i) = z_i \text{ for every } i = 1, \dots, k-1 \text{ and } \exists z_k :\ h(x_k) = z_k]$
$= \sum_{z_k \in M} P[h(x_i) = z_i \text{ for every } i = 1, \dots, k] \leq m\frac{1}{m^k}$
2. $P[h(x_1) = h(x_2)] = P[\exists z \in M :\ h(x_1) = z \text{ and } h(x_2) = z]$
$= \sum_{z \in M} P[h(x_1) = z \text{ and } h(x_2) = z] \leq m\frac{1}{m^2}$
3. Consider $\mathcal{H} = \{x \mapsto a;\ a \in M\}$. Then, $P[h_a(x) = z] = P[a = z] = \frac{1}{m}$ but $P[h_a(x_1) = h_a(x_2)] = P[a = a] = 1$.

Hash tables: Universal hashing: Multiply-mod-prime

Definition

- p is a prime greater than u
- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$
- $\mathcal{H} = \{h_{a,b};\ a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\}\}$

Lemma

For every prime p, let $[p] = \{0, \dots, p-1\}$. For every different $x_1, x_2 \in [p]$, equations

$$y_1 = ax_1 + b \mod p$$
$$y_2 = ax_2 + b \mod p$$

define a bijection between $(a, b) \in [p]^2$ and $(y_1, y_2) \in [p]^2$. ①
Furthermore, these equations define a bijection between $\{(a, b) \in [p]^2;\ a \neq 0\}$ and $\{(y_1, y_2) \in [p]^2;\ y_1 \neq y_2\}$. ②

Universality

The multiply-mod-prime set of functions $\mathcal{H}$ is universal. ③

1. Subtracting these equations, we get $a(x_1 - x_2) \equiv y_1 - y_2 \bmod p$. Hence, for given pair (y_1, y_2) there exists exactly one $a = (y_1 - y_2)(x_1 - x_2)^{-1}$ in the field GF(p). Similarly, there exists exactly one $b = y_1 - ax_1$ in the field GF(p).
2. Indeed, $y_1 = y_2$ if and only if $a = 0$.
3. For $x_1 \neq x_2$ we have a collision $h_{a,b}(x_1) = h_{a,b}(x_2)$ iff $y_1 \equiv y_2 \pmod m$. Note that $y_1 \neq y_2$. For given y_1 there are at most $\lceil \frac{p}{m} \rceil - 1$ values y_2 such that $y_1 \equiv y_2 \pmod m$ and $y_1 \neq y_2$. So, the total number of colliding pairs from $\{(y_1, y_2) \in [p]^2;\ y_1 \neq y_2\}$ is at most $p(\lceil \frac{p}{m} \rceil - 1) \leq p(\frac{p+m-1}{m} - 1) \leq \frac{p(p-1)}{m}$. The bijection implies that there are at most $\frac{p(p-1)}{m}$ pairs from $\{(a, b) \in [p]^2;\ a \neq 0\}$ causing a collision $h_{a,b}(x_1) = h_{a,b}(x_2)$. Hence, $P[h_{a,b}(x_1) = h_{a,b}(x_2)] \leq \frac{p(p-1)}{m|\mathcal{H}|} \leq \frac{1}{m}$.

Hash tables: Universal hashing: Multiply-mod-prime

Definition

- p is a prime greater than u
- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$
- $\mathcal{H} = \{h_{a,b};\ a \in \{0, \dots, p-1\}, b \in \{0, \dots, p-1\}\}$

Lemma

For every prime p, let $[p] = \{0, \dots, p-1\}$. For every different $x_1, x_2 \in [p]$, equations

$$y_1 = ax_1 + b \mod p$$
$$y_2 = ax_2 + b \mod p$$

define a bijection between $(a, b) \in [p]^2$ and $(y_1, y_2) \in [p]^2$.

2-universality

For every $x_1, x_2 \in U$, $x_1 \neq x_2$, and $y_1, y_2 \in M$ it holds

$$P[h_{a,b}(x_1) = z_1 \text{ and } h_{a,b}(x_1) = z_2] \leq \frac{\lceil \frac{p}{m} \rceil^2}{p^2}$$

So, the multiply-mod-prime set of functions $\mathcal{H}$ is not 2-universal. ① ②

1. There are at most $\lceil \frac{p}{m} \rceil^2$ pairs (y_1, y_2) such that $z_1 = y_1 \mod m$ and $z_2 = y_2 \mod m$. The bijection implies that there are at most $\lceil \frac{p}{m} \rceil^2$ pairs (a, b) such that $h_{a,b}(x_1) = z_1$ and $h_{a,b}(x_2) = z_2$.
2. Considering $a \in \{1, \dots, p-1\}$ leads to probability

$$P[h_{a,b}(x_1) = z_1 \text{ and } h_{a,b}(x_1) = z_2] \leq \frac{\lceil \frac{p}{m} \rceil^2}{p(p-1)}.$$

Hash tables: Universal hashing: Multiply-shift

Bits selection

For positive integers a, b, x, let $\text{bit}_{a,b}(x) = \left\lfloor \frac{x \bmod 2^b}{2^a} \right\rfloor$.

Multiply-shift

- Assume $u = 2^w$ and $m = 2^l$
- $h_a(x) = \text{bit}_{w-l,w}(ax)$
- $\mathcal{H} = \{h_a;\ a \text{ odd } w\text{-bit integer}\}$

Example is C

```
uint64_t hash(uint64_t x, uint64_t l, uint64_t a)
{ return (a*x) >> (64-l); }
```

Universality (without a proof)

For every $x_1, x_2 \in [2^w]$, $x_1 \neq x_2$ it holds $P[h_a(x_1) = h_a(x_2)] \leq \frac{2}{m}$.

Hash tables: Universal hashing: Multiply-shift

Bits selection

For positive integers a, b, x, let $\text{bit}_{a,b}(x) = \left\lfloor \frac{x \bmod 2^b}{2^a} \right\rfloor$.

Multiply-shift

- Assume $u = 2^w$ and $m = 2^l$ and $v \geq w + l - 1$.
- $h_{a,b}(x) = \text{bit}_{v-l,v}(ax + b)$
- $\mathcal{H} = \{h_{a,b};\ a, b \in [2^v]\}$

Lemma

If α and β are relatively prime, then $x \mapsto \alpha x \bmod \beta$ is a bijection on $[\beta]$. ①

2-universality

$\mathcal{H}$ is 2-universal, that is or every $x_1, x_2 \in [2^w]$, $x_1 \neq x_2$ and $z_1, z_2 \in M$ it holds $P[h(x_1) = z_1 \text{ and } h(x_2) = z_2] \leq \frac{1}{m^2}$.

1. Consider $x_1, x_2, y \in [\beta]$ such that $y \equiv \alpha x_1 \equiv \alpha x_2 \pmod \beta$. Then, β divides $\alpha(x_2 - x_1)$. Since α and β are relatively prime, β divides $x_2 - x_1$ which implies $x_1 = x_2$.

Hash tables: Universal hashing: Multiply-shift: 2-universality

$\mathcal{H} = \{x \mapsto \text{bit}_{v-l,v}(ax + b);\ a, b \in [2^v]\}$ is 2-universal where $v \geq w + l - 1$

1. Let s be the index of the least significant 1-bit in $(x_2 - x_1)$
2. Let o be the odd number such that $x_2 - x_1 = o2^s$
3. $a \mapsto ao \bmod 2^v = \text{bit}_{0,v}(ao)$ is a bijection on $[2^v]$ ①
4. $a \mapsto \text{bit}_{s,v+s}(ao2^s) = \text{bit}_{s,v+s}(a(x_2 - x_1))$ is a bijection on $[2^v]$
5. $a \mapsto \text{bit}_{s,v}(a(x_2 - x_1))$ is a 2^s-to-1 mapping $[2^v] \to [2^{v-s}]$
6. $b \mapsto \text{bit}_{s,v}(ax_1 + b2^s)$ is a bijection on $[2^{v-s}]$ for every $a \in [2^v]$
7. $b \mapsto \text{bit}_{s,v}(ax_1 + b)$ is a 2^s-to-1 mapping $[2^v] \to [2^{v-s}]$ for every $a \in [2^v]$
8. $(a, b) \mapsto (\text{bit}_{s,v}(ax_1 + b), \text{bit}_{s,v}(a(x_2 - x_1)))$ is a 2^{2s}-to-1 mapping $[2^v]^2 \to [2^{v-s}]^2$
9. $\text{bit}_{s,\infty}(ax_2 + b) = \text{bit}_{s,\infty}((ax_1 + b) + a(x_2 - x_1)) = \text{bit}_{s,\infty}(ax_1 + b) + \text{bit}_{s,\infty}(a(x_2 - x_1))$ ②
10. $(a, b) \mapsto (\text{bit}_{s,v}(ax_1 + b), \text{bit}_{s,v}(ax_2 + b))$ is a 2^{2s}-to-1 mapping $[2^v]^2 \to [2^{v-s}]^2$ ③
11. Since $w + l - 1 \leq v$ and $s < w$, it follows that $s \leq w - 1 \leq v - l$
12. $(a, b) \mapsto (\text{bit}_{v-l,v}(ax_1 + b), \text{bit}_{v-l,v}(ax_2 + b))$ is a $2^{2(v-l)}$-to-1 mapping $[2^v]^2 \to [2^l]^2$
13. If a, b are independently uniformly distributed on $[2^v]$, then $P[h(x_1) = z_1 \text{ and } h(x_2) = z_2] = \frac{2^{2(v-l)}}{2^{2v}} = \frac{1}{m^2}$.

1. Follows from lemma for $\alpha = o$ and $\beta = 2^v$.
2. The second equality uses $\text{bit}_{0,s}(a(x_2 - x_1)) = 0$.
3. Since $(\alpha, \beta) \mapsto (\alpha, \alpha + \beta)$ is a bijection.

Hash tables: Universal hashing: Multiply-shift for vectors

Multiply-shift for fix-length vectors

- Hash a vector $x_1, \dots, x_d \in U = [2^w]$ into $S = [2^l]$ and let $v \geq w + l - 1$
- $h_{a_1,\dots,a_d,b}(x_1, \dots, x_d) = \text{bit}_{v-l,v}(b + \sum_{i=1}^d a_i x_i)$
- $\mathcal{H} = \left\{h_{a_1,\dots,a_d,b};\ a_1, \dots, a_d, b \in [2^v]\right\}$
- $\mathcal{H}$ is 2-universal (without a proof)

Multiply-shift for variable-length string

- Hash a string $x_0, \dots, x_d \in U$ into $[p]$ where $p \geq u$ is a prime.
- $h_a(x_0, \dots, x_d) = \sum_{i=0}^d x_i a^i \bmod p$ ①
- $\mathcal{H} = \{h_a;\ a \in [p]\}$
- $P[h_a(x_0, \dots, x_d) = h_a(x'_0, \dots, x'_{d'})] \leq \frac{d+1}{p}$ for two different strings with $d' \leq d$. ②

Multiply-shift for variable-length string II

- Hash a string $x_0, \dots, x_d \in U$ into $[m]$ where $p \geq m$ is a prime.
- $h_{a,b,c}(x_0, \dots, x_d) = \left(b + c\sum_{i=0}^d x_i a^i \bmod p\right) \bmod m$
- $\mathcal{H} = \{h_{a,b,c};\ a, b, c \in [p]\}$
- $P[h_{a,b,c}(x_0, \dots, x_d) = h_{a,b,c}(x'_0, \dots, x'_{d'})] \leq \frac{2}{p}$ for different strings with $d' \leq d \leq \frac{p}{m}$.

1. $x_0, \dots, x_d$ are coefficients of a polynomial of degree d.
2. Two different polynomials of degree at most d have at most $d + 1$ common points, so there are at most $d + 1$ colliding values α.

Hash tables: Universal hashing: Multiply-shift for vectors

Tabulation hashing

- Random tabular $T_1, \dots T_d$
- $T_1(x_1) \oplus \dots \oplus T_d(x_d)$

Mersenne prime

- Prime in the form $p = 2^a - 1$ is called Mersenne prime.
- E.g. $2^2 - 1$, $2^3 - 1$, $2^{31} - 1$, $2^{61} - 1$, $2^{89} - 1$, $2^{107} - 1$
- $x \equiv (x\&p) + (x >> a) \pmod p$

Outline

Geometry

Types of problems

- Given set S of points (or other geometrical objects) in $\mathbb{R}^d$.
- Find the nearest point of S for a given point.
- Find all points of S which lie in a given region, e.g. d-dimensional rectangle.

Nearest point in $\mathbb{R}^1$

Given set of points S in $\mathbb{R}$ of size n, find the nearest point of S to a given point x.

Static Array: query in $\mathcal{O}(\log n)$

Dynamic Balanced search tree: query and update in $\mathcal{O}(\log n)$

Range query in $\mathbb{R}^1$

Given a finite set of points S in $\mathbb{R}$, find all points of S in a given interval $\langle a, b\rangle$ where k is the number of points in the interval.

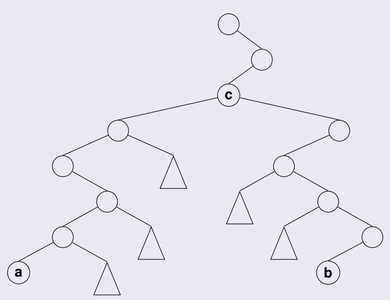
Static Array: query in $\mathcal{O}(k + \log n)$

Dynamic Balanced search tree: query and update in $\mathcal{O}(k + \log n)$

Geometry: Range query in $\mathbb{R}^1$

Example of $1D$ range tree

For simplicity, consider a binary search tree containing points only in leaves.



Drawn subtrees contain points exactly points between a and b. ①
How to determine the number of points in a given interval in $\mathcal{O}(\log n)$? ②

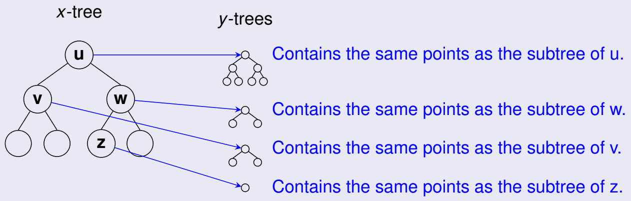
1. Nodes a and b are actually the successor of a and the predecessor of b, respectively.
2. Remember the number of leaves in every subtree.

Geometry: 2D range trees

Description

- Search tree for x-coordinates with points in leaves (x-tree).
- Every inner node u contains in its subtree of all points $S_u \subset S$ with x-coordinate in some interval.
- Furthermore, the inner node u also contains a search tree of points S_u ordered by y-coordinates (y-tree).

Example



Et cetera.

Geometry: 2D range trees: Space complexity

Vertical point of view

Every point p stored in exactly one leaf l of the x-tree; and moreover, p is also stored in all y-trees corresponding to all nodes on the path from the x-root to l.

Horizontal point of view

Every level of x-tree decomposes the set of points by x-coordinates. Therefore, y-trees corresponding to one level of x-tree contain every point exactly once.

Space complexity

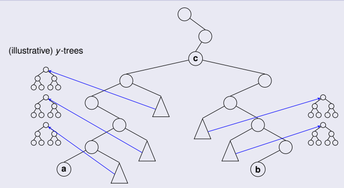
Since every point is stored in $\mathcal{O}(\log n)$ y-trees, the space complexity is $\mathcal{O}(n \log n)$.

Geometry: 2D range trees: Range query

Range query

1. Search for keys a_x and b_x in the x-tree.
2. Identify all inner nodes in the x-tree which store points with x-coordinate in the interval $\langle a_x, b_x \rangle$.
3. Run $\langle a_y, b_y \rangle$-query in all corresponding y-trees.

Example



Complexity

$\mathcal{O}\left(k + \log^2 n\right)$, since $\langle a_y, b_y \rangle$-query is run in $\mathcal{O}(\log n)$ y-trees.

Geometry: 2D range trees: Build

Straightforward approach

Create x-tree and then all y-trees using operation insert. Complexity is $\mathcal{O}\left(n \log^2 n\right)$.

Faster approach

First, create two arrays of points sorted by x and y coordinates. Then, recursively …

1. Let x-root be the medium of all points by x-coordinate.
2. Create y-tree for the x-root. ①
3. Split both sorted arrays by x-root.
4. Recursively create both children of x-root.

Complexity

- Recurrence formula $T(n) = 2T(n/2) + \mathcal{O}(n)$ ②
- Complexity is $\mathcal{O}(n \log n)$.

1. Given an array of sorted elements, most balanced search trees can be built in $\mathcal{O}(n)$-time.
2. Use master theorem, or observe that building one level of x-tree takes $\mathcal{O}(n)$-time.

Geometry: Higher dimensional range trees

3D range trees

1. Create 2D range tree for x and y coordinates.
2. For every node u in every y-tree, create a search tree ordered z-coordinate containing all points of the subtree of u.

d-dimensional range trees

Add dimensions one by one likewise in 3D range tree.

Complexity ①

Space: $\mathcal{O}\left(n \log^{d-1} n\right)$ since every point is stored in $\mathcal{O}\left(\log^2 n\right)$ z-trees, etc.

Query: $\mathcal{O}\left(k + \log^d n\right)$ since $\langle a_z, b_z \rangle$-query is run in $\mathcal{O}\left(\log^2 n\right)$ z-trees, etc.

Build: $\mathcal{O}\left(n \log^d n\right)$ if dimension-trees are created one-by-one by insertion.

$\mathcal{O}\left(n \log^{d-1} n\right)$ if we use the faster approach likewise in 2D.

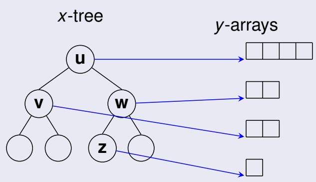
1. Dimension d is assumed to be a fix parameter.

Geometry: Layered range trees

2D case

Replace y-trees by sorted arrays.

Example



Et cetera.

Higher dimension

Replace trees of the last dimension by sorted arrays.

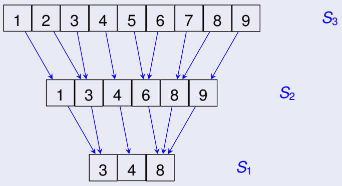
Geometry: Fractional cascading

Motivative problem

Given sets $S_1 \subseteq \cdots \subseteq S_m$ where $|S_m| = n$, create a data structure for fast searching elements $x \in S_1$ in all sets $S_1, \ldots, S_m$. ①

Fractional cascading

Every set S_i is sorted. Furthermore, every element in the array of S_i has a pointer to the same element in S_{i-1}. ②



Complexity of a search in m sets

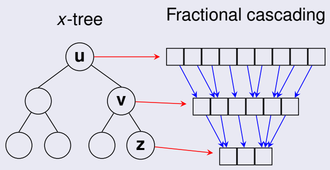
$\mathcal{O}(m + \log n)$

1. A straightforward solution gives complexity $\mathcal{O}(m \log n)$.
2. Elements $S_i \setminus S_{i-1}$ point to their predecessors or successors.

Geometry: Layered range trees and fractional cascading

Using fractional cascading

Use fractional cascading for the last dimension arrays, e.g. $d = 2$:



Complexity of one range query in 2D

- Search in the x-tree takes $\mathcal{O}(\log n)$.
- Binary search for a_y and b_y in y-arrays takes $\mathcal{O}(\log n)$.

Complexity of one range query in d dimensions

$\mathcal{O}\left(k + \log^{d-1} n\right)$

Geometry: Intermezzo: Weight balanced trees: BB[α]-tree

Description (Jürg Nievergelt, Edward M. Reingold [5])

A binary search tree is BB[α]-tree if for every node u

- $s_{u.left} \geq \alpha s_u - 1$ and
- $s_{u.right} \geq \alpha s_u - 1$

where the size s_u is the number of leaves in the subtree of u. ①

Height

The height of a BB[α]-tree is at most $\log_{\frac{1}{1-\alpha}}(n) + \mathcal{O}(1) = \mathcal{O}(\log n)$.

Balancing after operations Insert and Delete

When a node u violates the weight condition, rebuild whole subtree in time $\mathcal{O}(s_u)$. ②

Amortized cost

- Another rebuild of a node u occurs after $\Omega(s_u)$ updates in the subtree of u.
- Therefore, amortized cost of rebuilding subtree is $\mathcal{O}(1)$, and
- update contributes to amortized costs of all nodes on the path from the root to leaf.

The amortized cost of operations Insert and Delete is $\mathcal{O}(\log n)$. ③

1. Clearly, $0 < \alpha < \frac{1}{2}$. The term "-1" is important only for small subtrees when their size is odd.
2. It is possible to use rotations to keep the BB[α]-tree balanced. However in range trees, a rotation in the x-tree leads to rebuilding many y-trees.
3. This proof can be directly reformulated into potential method as follows. We define a potential $\Phi(u)$ of a node u to be

$$\Phi(u) = \begin{cases} 0 & \text{if } s_{u.left} = s_{u.right} = \frac{s_u}{2} \\ s_u & \text{if } \min\{s_{u.left}, s_{u.right}\} = \alpha s_u \end{cases}$$

and all other cases are defined using the linear interpolation of these two cases, that is

$$\Phi(u) = \frac{1}{1-2\alpha}\left(s_u - 2\min\{s_{u.left}, s_{u.right}\}\right).$$

This potential gives enough money when reconstruction is needed and zero after the reconstruction. Observe that the change of potential $\Delta\Phi(u)$ is at most $\mathcal{O}(1)$ when an element is inserted or deleted in the subtree of u. The total potential Φ is the sum of potentials of all nodes and its change is at most $\mathcal{O}(\log n)$ for an operation Insert or Delete (excluding reconstruction).

Geometry: Range trees using BB[α]-trees

Dynamic range trees

- For simplicity, consider BB[α]-tree for every dimension including the last one. ①
- Rotations in range trees are hard.
- However, reconstruction of a (sub)tree on n points takes $\mathcal{O}\left(n \log^{d-1} n\right)$. ②

2D case

- Reconstruction in the y-subtree of a node u takes $\mathcal{O}(s_u)$ time and another reconstruction occurs after $\Omega(s_u)$ updates in the y-subtree of u, so the amortized cost of rebuilding one y-subtree is $\mathcal{O}(1)$.
- Reconstruction in the x-subtree of a node u and following y-trees takes $\mathcal{O}(s_u \log s_u)$ time and another reconstruction occurs after $\Omega(s_u)$ updates time in the x-subtree of u, so the amortized cost of rebuilding one x-subtree is $\mathcal{O}(\log s_u)$.
- One update contributes to amortized costs in $\Omega(\log n)$ x-subtrees and $\Omega(\log^2 n)$ y-trees.
- Amortized cost of operations Insert and Delete is $\mathcal{O}\left(\log^2 n\right)$.

1. Without fractional cascading.
2. Balancing the x-tree requires reconstruction of trees in all dimensions.

Geometry: Range trees using BB[α]-trees

3D case

- Reconstruction in the z-subtree of a node u takes $\mathcal{O}(s_u)$ time and another reconstruction occurs after $\Omega(s_u)$ updates in the y-subtree of u, so the amortized cost of rebuilding one y-subtree is $\mathcal{O}(1)$.
- Reconstruction in the y-subtree of a node u and following z-trees takes $\mathcal{O}(s_u \log s_u)$ time and another reconstruction occurs after $\Omega(s_u)$ updates time in the y-subtree of u, so the amortized cost of rebuilding one y-subtree is $\mathcal{O}(\log s_u)$.
- Reconstruction in the x-subtree of a node u and following y-trees and z-trees takes $\mathcal{O}\left(s_u \log^2 s_u\right)$ time and another reconstruction occurs after $\Omega(s_u)$ updates time in the x-subtree of u, so the amortized cost of rebuilding one x-subtree is $\mathcal{O}\left(\log^2 s_u\right)$.
- One update contributes to amortized costs in $\Omega(\log n)$ x-subtrees and $\Omega(\log^2 n)$ y-trees and $\Omega(\log^3 n)$ z-trees.
- Amortized cost of operations Insert and Delete is $\mathcal{O}\left(\log^3 n\right)$.

d-dimensional range trees using BB[α]-trees

- Range query in $\mathcal{O}\left(k + \log^d n\right)$ worst case. ①
- Insert and Delete in $\mathcal{O}\left(\log^d n\right)$ amortized cost. ②

1. When we apply fractional cascading on leaves of a tree instead of arrays, we obtain query in $\mathcal{O}\left(k + \log^{d-1} n\right)$ without changing the complexity for updates.
2. The actual time for m updates is $\mathcal{O}\left(n \log^{d-1} n + m \log^{d} n\right)$.

Geometry: Range trees: Further improvements

Bernard Chazelle [1, 2]

d-dimensional range query in $\mathcal{O}\left(k + \log^{d-1} n\right)$ time and $\mathcal{O}\left(n \left(\frac{\log n}{\log \log n}\right)^{d-1}\right)$ space.

Bernard Chazelle, Leonidas J. Guibas [3]

d-dimensional range query in $\mathcal{O}\left(k + \log^{d-2} n\right)$ time and $\mathcal{O}\left(n \log^{d} n\right)$ space.

Geometry: Interval trees

Input

Set of intervals $S = \{I_1, \dots, I_n\}$ where $I_i = \langle a_i, b_i \rangle$.

Recursive contruction of interval trees

Interval tree is a binary tree. Let

- m be the medium of $2n$ endpoints $a_1, b_1, \dots, a_n, b_n$,
- $S_m = \{I_i;\ a_i \leq m \leq b_i\}$ be intervals containing m,
- $S_l = \{I_i;\ b_i < m\}$ be intervals smaller than m and
- $S_r = \{I_i;\ m < a_i\}$ be intervals greater than m.

The root of the tree contains

- two arrays of intervals S_m sorted by left and right end-points,
- interval trees for intervals S_l as the left child and
- interval trees for intervals S_r as the right child. ①

Complexity

- Time complexity for construction is $\mathcal{O}(n \log n)$. ②
- Space complexity is $\mathcal{O}(n)$. ③

1. If S_l or S_r is empty, then there is no left or right child, respectively.
2. There are at most n end-points smaller than m, so S_l contains at most $\frac{n}{2}$ intervals. Therefore, the time complexity satisfies the recurrence formula $T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$.
3. Every interval is stored in exactly one node. If S_m is empty, then n is even and both S_l and S_r contains $\frac{n}{2}$ intervals. There are at most $n-1$ such nodes. Therefore, the tree has at most $2n-1$ nodes.

Geometry: Interval trees: Intersection interval query

Problem description

Given query interval $Q = \langle a_q, b_q \rangle$, find all intervals intersecting with Q.

Recursive algorithm

```
if a_q ≤ m ≤ b_q then
    Write all intervals S_m
    Recursively process both children
else if b_q < m then
    Use the array of intervals S_m sorted by left end-points to find all intervals of S_m
    intersecting with Q
    Recursively process the left child
else
    Use the array of intervals S_m sorted by right end-points to find all intervals of S_m
    intersecting with Q
    Recursively process the right child
```

Complexity

$\mathcal{O}(k + \log n)$

Geometry: Segment trees

Input

Set of intervals $S = \{I_1, \dots, I_n\}$ where $I_i = \langle a_i, b_i \rangle$.

Query

Given point p, find all intervals of S containing p.

Trivial approach

1. Let $x_1, \dots, x_m$ be sorted end-points $\{a_1, b_1, \dots, a_n, b_n\}$ without duplicities.
2. Split $\mathbb{R}$ into blocks $(-\infty, x_1)$, $\{x_1\}$, (x_1, x_2), $\{x_2\}$, $\dots$, $\{x_m\}$, (x_m, ∞).
3. For every block, store all intervals of S containing the block.

Complexity

- Time for query: $\mathcal{O}(k + \log n)$
- Time for construction: $\mathcal{O}(n^2)$
- Space: $\mathcal{O}(n^2)$

Useful only for counting queries where every block contains the number of intervals.

Geometry: Segment trees

Idea of segment trees

- Let blocks $(-\infty, x_1)$, $\{x_1\}$, $\dots$, (x_m, ∞) be leaves of a binary tree.
- Every node stores the union of all blocks in its subtree.
- If two siblings store the same interval, store the interval in their parent instead.
- In the query, walk from the root to a leaf with a block containing a given point and print all intervals stored in all nodes on the path.

Space complexity

Every interval is stored in at most two nodes of every level of the tree.
Therefore, space complexity is $\mathcal{O}(n \log n)$.

Time complexity of a construction

First, sort all end-points and create the binary tree. Then, add all intervals using a top-down recursion.
Therefore, time complexity is $\mathcal{O}(n \log n)$.

Time complexity of a query

$\mathcal{O}(k + \log n)$.

Geometry: Priority search tree

Heap and search tree in one binary tree

If every element e has a key $e.key$ and a priority $e.priority$, is it possible to store a set of elements in a binary tree so that

- the min-heap property is satisfied for priorities and
- the search-tree property is satisfied for keys? ①

Relax the search tree property

Priority search tree is a binary tree having one element in every node so that

- the min-heap property is satisfied for priorities and
- elements can be fount by their keys in $\mathcal{O}(\log n)$ time.

Top-down recursive construction of a priority search tree

The root of the priority search tree storing a set of elements S contains

- the element e of S with the smallest priority,
- the median key m of all elements of S, ②
- the left subtree stores all elements with keys smaller than m (except e) and
- the right subtree stores all elements with keys greater than m (except e). ③

1. Observe that if all keys and all priorities are pair-wise different, then there exists a unique binary tree storing all elements.
2. Note that m is not the key of the element e (unless e coincidently has the median key).
3. Observe that this tree does not satisfies the search-tree condition in general.

Geometry: Priority search tree

Complexity

- Space complexity is $\mathcal{O}(n)$
- Construction in $\mathcal{O}(n \log n)$-time
- Find the element with the smallest priority in $\mathcal{O}(1)$-time
- Find the element with a given key in $\mathcal{O}(\log n)$-time
- Delete the element with the smallest priority in $\mathcal{O}(\log n)$-time ①

Applications

- Find the element with key in a given range and the smallest priority.
- Grounded 2D range search problem: Given a set of points in $\mathbb{R}^2$, find points in the range $\langle a_x, b_x \rangle \times (-\infty, b_y \rangle$.

1. After a deletion, nodes do not store the median keys of their subtree. Although the height of the tree is not increased by an operation delete, the tree may degenerate.

Outline

[1] Bernard Chazelle.
Lower bounds for orthogonal range searching: I. the reporting case.
Journal of the ACM (JACM), 37(2):200–212, 1990.

[2] Bernard Chazelle.
Lower bounds for orthogonal range searching: part ii. the arithmetic model.
Journal of the ACM (JACM), 37(3):439–463, 1990.

[3] Bernard Chazelle and Leonidas J Guibas.
Fractional cascading: I. a data structuring technique.
Algorithmica, 1(1-4):133–162, 1986.

[4] Donald Ervin Knuth.
Notes on "open" addressing.
http://algo.inria.fr/AofA/Research/11-97.html, 1963.

[5] Jürg Nievergelt and Edward M Reingold.
Binary search trees of bounded balance.
SIAM journal on Computing, 2(1):33–43, 1973.

[6] Rasmus Pagh and Flemming Friche Rodler.
Cuckoo hashing.
Journal of Algorithms, 51(2):122–144, 2004.

[7] Mihai Patraşcu.
Better guarantees for chaining and linear probing.

http://infoweekly.blogspot.cz/2010/02/better-guarantees-for-chaining-and.html.
blogspot, February 2, 2010.