Data Structures 1

NTIN066

Jirka Fink

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

Winter semester 2017/18
Last change on January 11, 2018

Jirka Fink: Data Structures 1

General information

E-mail fink@ktiml.mff.cuni.cz
Homepage https://ktiml.mff.cuni.cz/˜fink/
Consultations Thursday, 17:15, office S305

Examination

- Implement given data structures
- Pass the exam

Content

Literatura

- 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

Amortized complexity

Motivation

- Consider a data structure which is usually very fast
- However in rare cases, it needs to reorganize its internal structure
- So, the worst-case complexity is quite slow
- This data structure may used by an algorithm
- We are interested in the total complexity or average complexity of many operations

Incrementing binary counter

Problem description

- Consider an n-bit binary counter with an arbitrary initial value
- Operation Increment changes the last false bit to true and all following bits to false
- The number changed bits is at most n
- What is the maximal number of bits changed during k operations Increment?

Aggregate analysis

- The last bit is changed during every operation; i.e. k-times
- The last but one bit is changed during every other operation; i.e. at most $\lceil k/2 \rceil$-times
- The i-th least significant bit is changed during every 2^i-th operation; i.e. at most $\lceil k/2^i \rceil$-times
- The number of changed bits during k operations Increment is at most $\sum_{i=0}^{n-1} \lceil k/2^i \rceil \leq \sum_{i=0}^{n-1} (1 + k/2^i) \leq n + k \sum_{i=0}^{n-1} 1/2^i \leq n + 2k$

Incrementing binary counter

Accounting method

- Change of one bit costs one coin
- We keep one coin on every true bit, so the initialization costs at most n coins
- For the operation Increment, changing true bits to false is pre-payed and we pay the change of the last false bit and its pre-payment, so one operation costs 2 coins
- The total cost of k operations is at most $n + 2k$

Potential method

- The potential of a false bit is 0 and the potential of a true bit is 1
- The potential of the counter is the sum of potentials of all bits
- Let T_i be the number of changed bits during i-th operation and Φ_i be the potential of the counter after i-th operation
- Observe that $T_i + \Phi_i - \Phi_{i-1} \leq 2$
- The number of changed bit during k operations Increment is at most $\sum_{i=1}^{k} T_i \leq \sum_{i=1}^{k} (2 + \Phi_{i-1} - \Phi_i) \leq 2k + \Phi_0 - \Phi_k \leq 2k + n$ since $0 \leq \Phi_i \leq n$

Dynamic array

Problem description

- We have an array of length p storing n elements and we need to implement operations Insert and Delete
- If $n = p$ and an element has to be inserted, then the length of the array is doubled
- If $4n = p$ and an element has to be deleted, then the length of the array is halved
- What is the number of copied elements during k operations Insert and Delete?

Aggregated analysis

- Let k_i be the number of operations between $(i-1)$-th and i-th reallocation
- The first reallocation copies at most $n_0 + k_1$ elements where n_0 is the initial number of elements
- The i-th reallocation copies at most $2k_i$ elements for $i \geq 2$
- Every operation without reallocation copies at most 1 element
- The total number of copied elements is at most $k + (n_0 + k_1) + \sum_{i \geq 2} 2k_i \leq n_0 + 3k$

Dynamic array

Potential method

- Consider the potential
$$\Phi = \begin{cases} 0 & \text{if } p = 2n \\ n & \text{if } p = n \\ n & \text{if } p = 4n \end{cases}$$
and piece-wise linear function in other cases
- Explicitly,
$$\Phi = \begin{cases} 2n - p & \text{if } p \leq 2n \\ p/2 - n & \text{if } p \geq 2n \end{cases}$$
- Change of the potential without reallocation is $\Phi_i - \Phi_{i-1} \leq 2$ ①
- Let T_i be the number of elements copied during i-th operation
- Hence, $T_i + \Phi_i - \Phi_{i-1} \leq 3$
- The total number of copied elements during k operations is $\sum_{i=1}^k T_i \leq 3k + \Phi_0 - \Phi_k \leq 3k + n_0$

1. $$\Phi' - \Phi = \begin{cases} 2 & \text{Insert and } p \leq 2n \\ -2 & \text{Delete and } p \leq 2n \\ -1 & \text{Insert and } p \geq 2n \\ 1 & \text{Delete and } p \geq 2n \end{cases}$$

Amortized complexity

Average of the aggregated analysis

- The amortized complexity of an operation is the total time of k operations over k assuming that k is sufficiently large.
- For example, the amortized complexity of operations Insert and Delete in the dynamic array is $\frac{\sum_{i=1}^k T_i}{k} \leq \frac{3k+n_0}{k} \leq 4 = \mathcal{O}(1)$ assuming that $k \geq n_0$.

Potential method

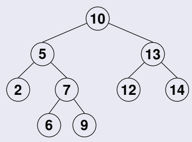
- Let Φ a potential which evaluates the internal representation of a data structure
- Let T_i be the actual time complexity of i-th operation
- Let Φ_i be the potential after i-th operation
- The amortized complexity of the operation is $\mathcal{O}(f(n))$ if $T_i + \Phi_i - \Phi_{i-1} \leq f(n)$ for every operation i in an arbitrary sequence of operations
- For example in dynamic array, $T_i + \Phi_i - \Phi_{i-1} \leq 3$, so the amortized complexity of operations Insert and Delete is $\mathcal{O}(1)$

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: $n - 1$

Weight balanced trees: BB[α]-tree

Description (Jürg Nievergelt, Edward M. Reingold [11])

- A binary search tree satisfying the following weight-balanced condition
- Let s_u be the number of nodes in a subtree of a node u ①
- For every node u it holds that the subtree of both children of u has at most αs_u nodes ②
- Clearly, make sense only for $\frac{1}{2} < \alpha < 1$

Height

- The subtree of every grandson of the root has at most $\alpha^2 n$ vertices
- The subtree of every node in the i-th level is at most $\alpha^i n$ vertices
- $\alpha^i n \geq 1$ only for $i \leq \log_{\frac{1}{\alpha}}(n)$
- The height of BB[α]-tree is $\Theta(\log n)$

Operation Build: Create BB[α]-tree from an array of sorted elements

- Create a root and set the middle element to be the key in the root
- Create both subtree of the root using recursing
- Time complexity is $\mathcal{O}(n)$

1. The node u is also counted in s_u.
2. Readers may find various variants conditions. The goal is to require that both children has similar number of nodes in their subtrees.

Weight balanced trees: BB[α]-tree

Operations Insert and Delete ①

- Insert/Delete given node similarly as in (non-balanced) binary search trees ②
- When a node u violates the weight condition, rebuild whole subtree in time $\mathcal{O}(s_u)$.

Amortized cost of rebalancing

- Between two consecutive rebuilds of a node u, there are at least $\Omega(s_u)$ updates in the subtree of u
- Therefore, amortized cost of rebuilding a subtree is $\mathcal{O}(1)$
- 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 $\mathcal{O}(\log n)$.

1. 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.
2. Complexity is $\mathcal{O}(\log n)$ since tree has height $\mathcal{O}(\log n)$.

Weight balanced trees: BB[α]-tree

Potential method

- The potencial of a node u is
$$\Phi(u) = \begin{cases} 0 & \text{if } |s_{l(u)} - s_{r(u)}| \leq 1 \\ |s_{l(u)} - s_{r(u)}| & \text{otherwise} \end{cases}$$
where $l(u)$ and $r(u)$ is left and right child of u, respectively
- The potential Φ of whole tree is the sum of potentials of all nodes
- Without reconstruction, Φ increases by $\mathcal{O}(\log(n))$
- If node u requires a reconstruction, then $\Phi(u) \geq \alpha s_u - (1-\alpha)s_u \geq (2\alpha - 1)s_u$
- So, the reconstruction is paid by the decreasement of the potential
- Every node in the new subtree has potential zero

Outline

Statically optimal tree

Goal

For a given sequence of operations FIND construct a binary search tree minimizing the total search time.

Formally

Consider elements $x_1, \dots, x_n$ with weights $w_1, \dots, w_n$. The cost of a tree is $\sum_{i=1}^n w_i h_i$ where h_i is the depth of an element x_i. The statically optimal tree is a binary search with minimal cost.

Construction (Exercise)

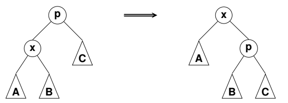
- $\mathcal{O}(n^3)$ – straightforward dynamic programming
- $\mathcal{O}(n^2)$ – improved dynamic programming (Knuth, 1971 [9])

What we can do if the search sequence is unknown?

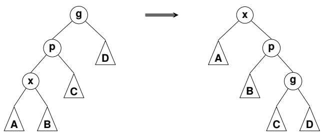
- Using rotations we keep recently searched elements closed to the root
- Operation SPLAY "rotates" a given element to the root
- Operation FIND finds a given element and calls operation Splay

Splay tree (Sleator, Tarjan, 1985 [19]): Operation SPLAY of a node x

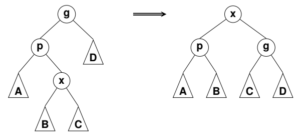
- Zig step: If the parent p of x is the root



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

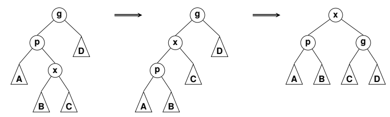


- Zig-zag step: x is a right child and p is a left child or vice versa

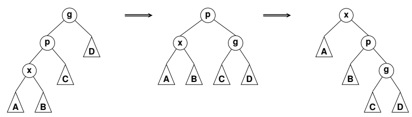


Splay tree: Operation SPLAY of a node x

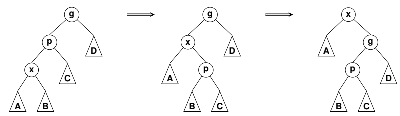
- Zig-zag step consists of two simple rotations of x with its parent



- Zig-zig step consists of two simple rotations,



- however two simple rotations of x with its parent lead to a different tree



Splay trees: Amortized analysis of operation SPLAY

Lemma

For $a, b, c \in \mathbb{R}^+$ satisfying $a + b \leq c$ it holds that $\log_2(a) + \log_2(b) \leq 2\log_2(c) - 2$.

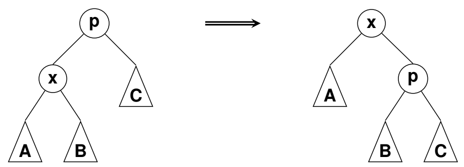
Proof

- Observe that $4ab = (a+b)^2 - (a-b)^2$
- From $(a-b)^2 \geq 0$ and $a + b \leq c$ it follows that $4ab \leq c^2$
- Applying the logarithm we obtain $\log_2(4) + \log_2(a) + \log_2(b) \leq \log_2(c^2)$

Notation

- Size $s(x)$ is the number of nodes in the subtree rooted at node x (including x)
- Potential of a node x is $\Phi(x) = \log_2(s(x))$
- Potential Φ is the sum of potentials of all nodes in the tree
- First, we will analyze the change of potential during a single step (rotation)
- Let s_i a Φ_i be sizes and potentials after the i-th step
- Let T_i be the number of rotations during one step

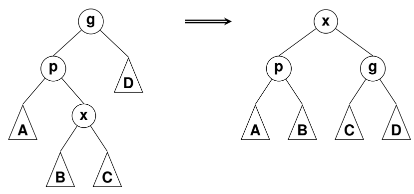
Splay trees: Zig step



Analysis

1. $\Phi_i(x) = \Phi_{i-1}(p)$
2. $\Phi_i(p) < \Phi_i(x)$
3. $\Phi_i(u) = \Phi_{i-1}(u)$ for every other nodes u
4. $$\begin{aligned} \Phi_i - \Phi_{i-1} &= \sum_{\text{nodes } u} (\Phi_i(u) - \Phi_{i-1}(u)) \\ &= \Phi_i(p) - \Phi_{i-1}(p) + \Phi_i(x) - \Phi_{i-1}(x) \\ &\leq \Phi_i(x) - \Phi_{i-1}(x) \end{aligned}$$
5. Hence, $\Phi_i - \Phi_{i-1} \leq \Phi_i(x) - \Phi_{i-1}(x)$

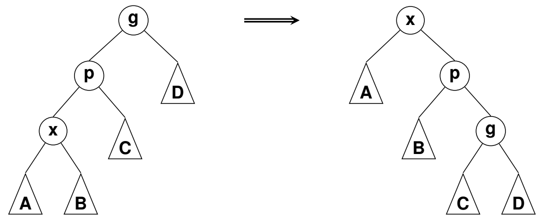
Splay trees: Zig-zag step



Analysis

1. $\Phi_i(x) = \Phi_{i-1}(g)$
2. $\Phi_{i-1}(x) < \Phi_{i-1}(p)$
3. $\Phi_i(p) + \Phi_i(g) \leq 2\Phi_i(x) - 2$
   - Observe that $s_i(p) + s_i(g) \leq s_i(x)$
   - From lemma it follows that $\log_2(s_i(p)) + \log_2(s_i(g)) \leq 2\log(s_i(x)) - 2$
4. $\Phi_i - \Phi_{i-1} = \Phi_i(g) - \Phi_{i-1}(g) + \Phi_i(p) - \Phi_{i-1}(p) + \Phi_i(x) - \Phi_{i-1}(x)$
   $\leq 2(\Phi_i(x) - \Phi_{i-1}(x)) - 2$

Splay trees: Zig-zig step



Analysis

1. $\Phi_i(x) = \Phi_{i-1}(g)$
2. $\Phi_{i-1}(x) < \Phi_{i-1}(p)$
3. $\Phi_i(p) < \Phi_i(x)$
4. $s_{i-1}(x) + s_i(g) \leq s_i(x)$
5. $\Phi_{i-1}(x) + \Phi_i(g) \leq 2\Phi_i(x) - 2$
6. $\Phi_i - \Phi_{i-1} = \Phi_i(g) - \Phi_{i-1}(g) + \Phi_i(p) - \Phi_{i-1}(p) + \Phi_i(x) - \Phi_{i-1}(x)$
   $\leq 3(\Phi_i(x) - \Phi_{i-1}(x)) - 2$

Splay trees: Analysis

Amortized complexity

- Zig-zig or zig-zag step:
  $T_i + \Phi_i - \Phi_{i-1} \leq 2 + 3(\Phi_i(x) - \Phi_{i-1}(x)) - 2 = 3(\Phi_i(x) - \Phi_{i-1}(x))$
- Zig step:
  $T_i + \Phi_i - \Phi_{i-1} \leq 1 + \Phi_i(x) - \Phi_{i-1}(x) \leq 1 + 3(\Phi_i(x) - \Phi_{i-1}(x))$
- Amortized complexity of one operation SPLAY (the sum of all steps):

$$\sum_{i\text{-th step}} (T_i + \Phi_i - \Phi_{i-1}) \leq 1 + \sum_{i\text{-th step}} 3(\Phi_i(x) - \Phi_{i-1}(x))$$
$$\leq 1 + 3(\Phi_{\text{last}}(x) - \Phi_0(x)) \quad (1)$$
$$\leq 1 + 3\log_2 n$$
$$= \mathcal{O}(\log n)$$

- Amortized complexity of operation SPLAY is $\mathcal{O}(\log n)$

Worst-case complexity of k operations SPLAY

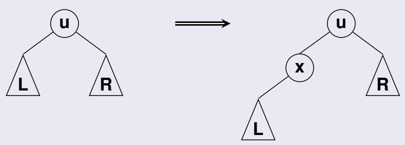
- Potential always satisfies $0 \leq \Phi \leq n \log_2 n$
- The difference between the final and the initial potential is at most $n \log_2 n$
- Complexity of k operations SPLAY is $\mathcal{O}((n + k)\log n)$

1. Zig step is used at most once during operation SPLAY, so we add "+1" once. After applying telescopic cancellation, only the initial $\Phi_0(x)$ and final $\Phi_{\text{last}}(x)$ potential remains. From the definition it follows that $\Phi_0(x) \geq 0$ and $\Phi_{\text{last}} = \log_2 n$.

Splay trees: Operation INSERT

INSERT a 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

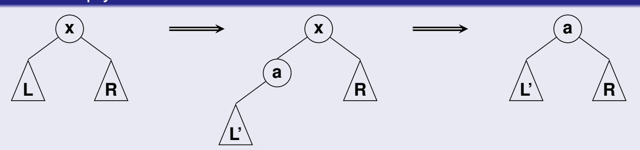
- Amortized complexity of FIND and SPLAY: $\mathcal{O}(\log n)$
- The potential Φ is increased by at most $\Phi(x) + \Phi(u) \leq 2\log n$
- Amortized complexity of operation INSERT is $\mathcal{O}(\log n)$

Splay trees: 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



The first homework: Splay trees

Overview of the problem

- Implement Splay tree with operations SPLAY, FIND, INSERT
- Implement "a naive" version, which uses simple rotations only
- Measure the average depth of search elements during operations FINDand SPLAY
- Study the dependency of the average depth on the size of a set of searched elements
- Study the average depth in a sequential test
- Deadline: October 29, 2017
- For a data generator and more details visit
  `https://ktiml.mff.cuni.cz/~fink/`

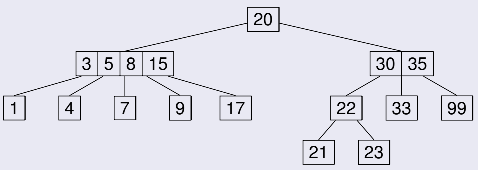
Outline

Search tree

Properties

- Inner nodes have arbitrary many children (usually at least 2)
- Inner node with k children has $k-1$ sorted keys
- The i-th key is greater than all keys in the i-th subtree and smaller than all keys in the $(i+1)$-th subtree for every key i
- Two ways of storing elements:
  - Elements are stored in leaves only
  - Elements are stored in all nodes (i.e. inner nodes contain element for every key)

Example



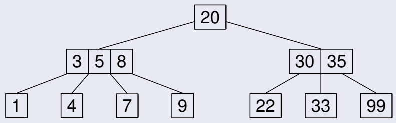
(a,b)-tree (Bayer, McCreight [1])

Properties

(a,b)-tree is a search tree satisfying the following properties

1. a, b are integers such that $a \geq 2$ and $b \geq 2a-1$
2. All internal nodes except the root have at least a and at most b children
3. The root has at most b children
4. All leaves are at the same depth
5. Elements are stored in leaves (simplifies the explanation)

Example: (2,4)-tree

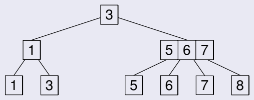


Operation FIND

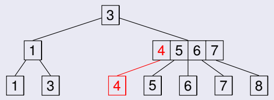
Search from the root using keys stored in internal nodes

(a,b)-tree: Operation INSERT

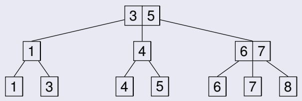
Insert 4 into the following (2,4)-tree



Add a new leaf into the proper parent



Recursively split node if needed



(a,b)-tree: Operation INSERT

Algorithm

```
Find the proper parent v of the inserted element
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 ← 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 ⌊(b+1)/2⌋ children of v to v' ① ②
    v ← u
```

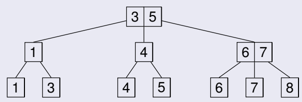
Time complexity

Linear in height of the tree

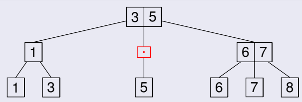
1. It is also necessary to update the list of keys and children of the parent u.
2. We should verify that we the resulting tree after the operation INSERT satisfies all properties required by the definition of (a,b)-tree. Indeed, we check that split nodes have at least a children (other conditions are trivial). A node requiring a split has $b+1$ children and it is split into two nodes with $\lfloor \frac{b+1}{2} \rfloor$ a $\lceil \frac{b+1}{2} \rceil$ children. Since $b \geq 2a-1$, each new node has at least $\lfloor \frac{b+1}{2} \rfloor \geq \lfloor \frac{2a-1+1}{2} \rfloor = \lfloor a \rfloor = a$ children as required.

(a,b)-tree: Operation DELETE

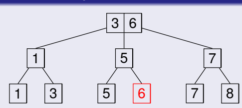
Delete 4 from the following (2,4)-tree

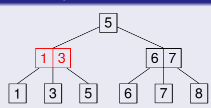


Find and delete the proper leaf



Recursively either share nodes from a sibling or fuse the parent





(a,b)-tree: Operation DELETE

Algorithm

```
Find the leaf l containing the deleted key
v ← parent of l
Delete l
while deg(v) < a and v is not the root do
    u ← 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 ← 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 of (a,b)-tree satisfies $\log_b n \leq d \leq 1 + \log_a n$.

Complexity

Time complexity of operations FIND, Insert and DELETE is $\mathcal{O}(\log n)$.

The number of modified nodes when (a,b)-tree is created using INSERT

- We create (a,b)-tree using operation INSERT
- We ask what the number of balancing operations (split) is ①
- Every split creates a new node
- A tree with n elements has at most n inner nodes
- The total number of splits is at most n
- The amortized number of modified nodes during one operation INSERT is $\mathcal{O}(1)$ ②

1. One balancing operation (node split) modifies a bounded number of nodes (split node, parent, children). Hence, the numbers of splits and modified nodes are asymptotically equal.
2. Note that operation INSERT has to find the proper leaf for the new element, so complexity of operation INSERT is $\mathcal{O}(\log n)$.

(a,b)-tree: Parallel access

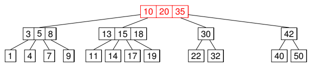
Goal
Efficient parallelization of operations FIND, INSERT and DELETE (assuming $b \geq 2a$).

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

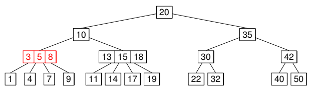
Operation DELETE
Update (move a child or merge with a sibling) every node with a children on path from the root to the deleted leaf.

(a,b)-tree: Parallel access: Example

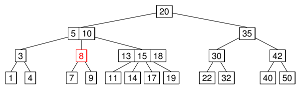
- INSERT element with a key 6 into the following (2,4)-tree



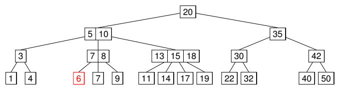
- First, we split the root



- Then, we continue to its left child which we also split



- Now, a node with key 8 does not require split and a new node can be added



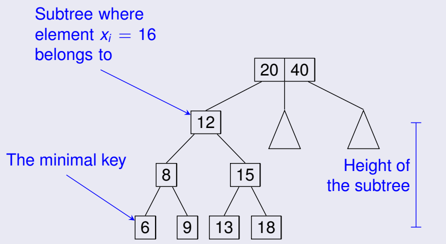
A-sort (Guibas, McCreight, Plass, Roberts [7])

Goals
Sort "almost" sorted list of elements

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

Example: INSERT element $x_i = 16$

- Walk from the leaf with minimal key to the root while x_i does not belong to the subtree of the current element
- INSERT x_i into the subtree where x_i belongs
- Height of that subtree is $\Theta(\log f_i)$, where f_i is the number of key smaller than x_i



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 # Elements are processed from the (almost) largest
do
  # Modified operation insert of x_i to T
  $v \leftarrow$ the leaf with the smallest key
  while *v is not the root and the smallest key stored in v's parent is greater than x_i* **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 elements

A-sort: Complexity

The inequality between 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
- Let $f_i = |\{j > i; x_j < x_i\}|$ be the number of keys smaller than x_i stored in the tree when x_i is inserted
- Let $F = \sum_{i=1}^n f_i$ be 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: $\mathcal{O}(n \log(F/n))$
  $\sum_i \log f_i = \log \prod_i f_i = n \log \sqrt[n]{\prod_i f_i} \leq n \log \frac{\sum_i f_i}{n} = n \log \frac{F}{n}$.
- 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 $\mathcal{O}(n \log\log n)$

(a,b)-tree: Applications

The number of modified nodes during a sequence of operations INSERT and DELETE (Huddleston, Mehlhorn, 1982 [8])
- Assume that $b \geq 2a$
- Consider a sequence of l operations INSERT and k operations DELETE
- The number of modified nodes is $\mathcal{O}(k + l + \log n)$
- The amortized number of modified nodes during INSERT or DELETE is $\mathcal{O}(1)$

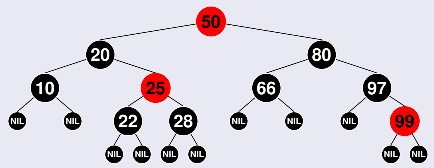
Similar data structures
- B-tree, B+ tree, B* tree
- 2-4-tree, 2-3-4-tree, etc.

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

Red-black tree: Definition

Definition
- Binary search tree with elements stored in inner nodes
- Every node is either red or black
- Paths from the root to all leaves contain the same number of black nodes
- Parent of a red node must be black
- Leaves are black ①

Example

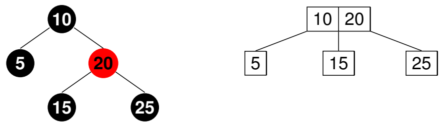


1. The last condition is not necessary but it simplifies operations. Note that we can also require that the root is black.

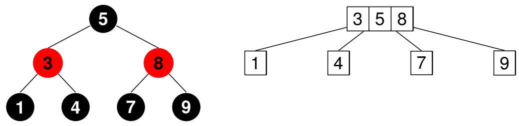
Red-black tree: Equivalence to (2,4)-tree

- A node with no red child

- A node with one red child [1]



- A node with two red children

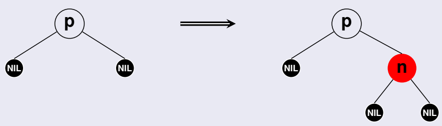


1. This equivalence is not unique. In this example, node with key 10 can also be red left child of a black node with key 20.

Red-black tree: Insert

Creating new node

- Find the position for the new element n and add it

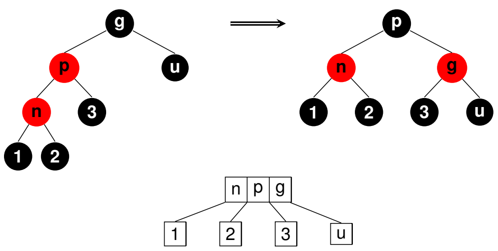


- If the parent p is red, balance the tree

Balancing

- Node n and its parent p are red. All other properties is satisfied.
- The grandparent g is black.
- The uncle u is red or black.

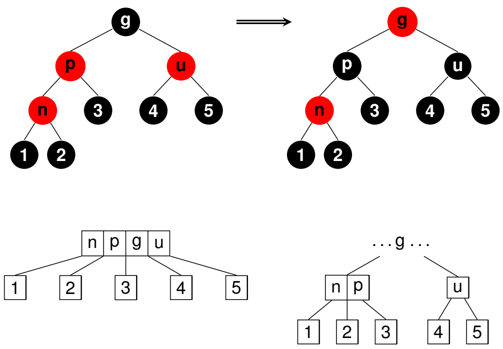
Red-black tree: INSERT — uncle is black



Note

The order of elements in (2,4)-tree depends on whether n is left or right child of p and whether p is left or right child of g.

Red-black tree: INSERT — uncle is red



Note

Splitting a node in (2,4)-tree moves the key g to the parent node which contains other keys and children. The last balancing operation has two cases.

Red-black tree: Properties

Corollary of the equivalence to (2,4)-tree

- Height of a red-black tree is $\Theta(\log n)$
- Complexity of operations FIND, INSERT and DELETEis $\mathcal{O}(\log n)$
- Amortized number of modified nodes during operations INSERT and DELETE is $\mathcal{O}(1)$
- Parallel access (top-down balancing)

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

Outline

Heap

Basic operations
- INSERT
- FINDMIN
- DELETEMIN
- DECREASE

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)

Properties
- An element has its priority which can be be decreased ①
- Elements stored in all nodes of a tree
- Priority of every node is always smaller than or equal than priorities of its children ②

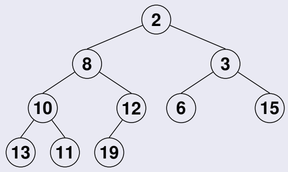
1. Since priority may change and it does not identify elements. Therefore in heaps, we use the work *priority* instead of *key*.
2. This condition implies that an element with the smallest priority is stored in the root, so it can be fount in time $\mathcal{O}(1)$.

Outline

d-regular heap

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

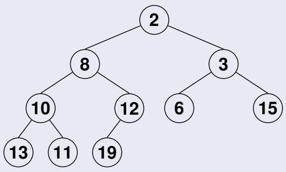
Example of a binary heap



Height
Height of a d-regular heap is $\Theta(\log_d n)$.

d-regular: 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 $2i+2$.

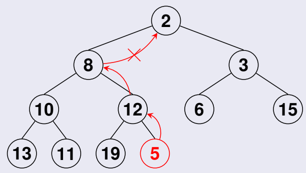
2	8	3	10	12	6	15	13	11	19

Parent

Children

d-regular heap: Operation INSERT

Example: Insert 5



2	8	3	10	12	6	15	13	11	19	5

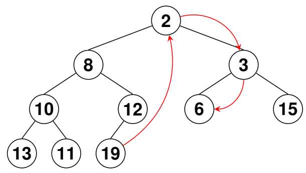
d-regular heap: Operations INSERT and DECREASE

Insert: Algorithm
Input: A new element with priority 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 priority greater than x* **do**
  Swap elements v and p
  $v \leftarrow p$

Operation DECREASE of a given node
Decrease priority and swap the element with parents when necessary (likewise in the operation INSERT).

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

d-regular heap: Operation DELETEMIN



Algorithm
Move the last element to the root v
while *Some children of v has smaller priority than v* **do**
  $u \leftarrow$ the child of v with the smallest priority
  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-regular heap: Building

Goal

Initialize a heap from a given array of elements

Algorithm

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

Correctness

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

d-regular heap: Building

Lemma

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

Complexity

1. Processing a node with a subtree of height h: $\mathcal{O}(dh)$
2. A complete subtree of height h has d^h leaves
3. Every leaf belong in at most one complete subtree of height h.
4. The number of nodes with a subtree of height h is at most $\frac{n}{d^h} + 1 \leq \frac{2n}{d^h}$ ①
5. The total time complexity is

$$\sum_{h=0}^{\lceil \log_d n \rceil} \frac{2n}{d^h} dh \leq 2nd \sum_{h=0}^{\infty} \frac{h}{d^h} = 2n \left(\frac{d}{d-1} \right)^2 \leq 2n2^2 = \mathcal{O}(n)$$

① We add “+1” to count at most one incomplete subtree.

Outline

Binomial tree

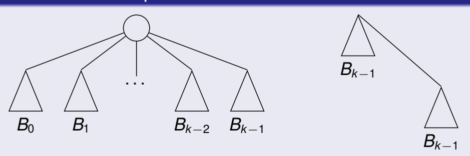
Definition

- 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

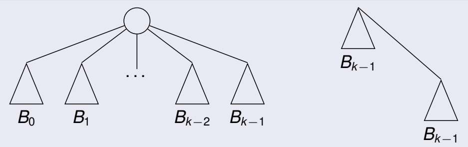
A binomial tree of order k is constructed from two binomial trees of order $k-1$ by attaching one of them as the rightmost child of the root of the other tree.

Recursions for binomial heaps

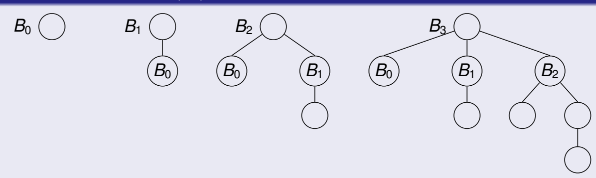


Binomial tree: Example

Recursions for binomial heaps

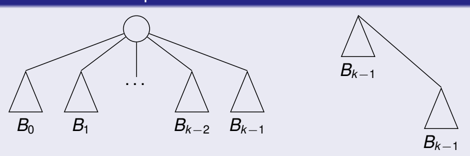


Binomial trees of order 0, 1, 2 and 3



Binomial tree: Properties

Recursions for binomial heaps



Observations

A binomial tree B_k of order k has

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

The subtree of a node with k children is isomorphic to B_k.

Set of binomial trees

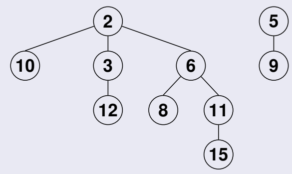
Observations

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

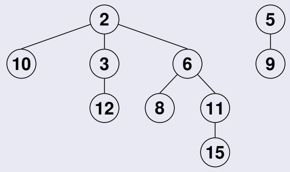
Binary number n	$=$	1	0	0	1	1	0	1	0
		$\downarrow$			$\downarrow$	$\downarrow$		$\downarrow$	
Binomial tree contains:		B_7			B_4	B_3		B_1	

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
- Every element is stored in one node of one tree.
- Each binomial tree obeys the minimum-heap property: the priority of a node is greater than or equal to the priority 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
		$\downarrow$			$\downarrow$	$\downarrow$		$\downarrow$	
Binomial tree contains:		B_7			B_4	B_3		B_1	

Binomial heap: Representation

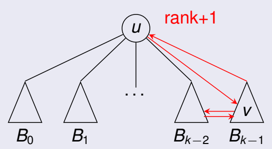
A node in a binomial tree contains
- an element (priority and value),
- a pointer to its parent,
- a pointer to its most-left and the most-right children, ①
- a pointer to its left and right sibling and ②
- the number of children (order).

Binomial heap
- Binomial trees are stored in a double linked list using pointers to siblings.
- Binomial heap keeps a pointer to an element of the smallest priority.

Trivial operations
- FINDMIN in $\mathcal{O}(1)$
- DECREASE: Similar as in regular heaps, complexity $\mathcal{O}(\log n)$

1. Actually, one pointer to any child is sufficient. In order to understand the analysis of Fibonacci heap, it is easier to assume that a parent has access to the first and the last children and new child is appended to the end of the list of children when two trees are joined.
2. Pointers for a double linked list of all children which is sorted by orders.

Binomial heap: Joining two heaps

Joining two binomial trees in time $\mathcal{O}(1)$


Joining two binomial heaps in time $\mathcal{O}(\log n)$
Join 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.

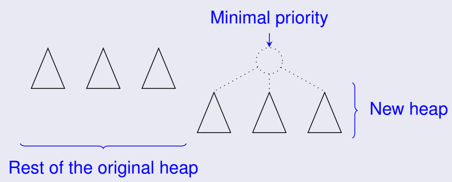
Example

Binomial trees	B_6	B_5	B_4	B_3	B_2	B_1	B_0
First heap	0	1	1	0	1	1	0
Second heap	0	1	1	0	1	0	0
Join	1	1	0	1	0	1	0

Binomial heap: Operations INSERTand DECREASE

Operation INSERT
- INSERTis implemented as join with a new tree of order zero containing new element.
- Complexity of INSERT is similar to the binary counter.
- The worst-case complexity is $\mathcal{O}(\log n)$.
- The amortized complexity is $\mathcal{O}(1)$.

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



Outline

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.

Operaions 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
- Find element with minimal priority

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 ⌈log₂(n+1)⌉
for each tree 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
```


Heap: Overview

Complexity table

	Binary	Binomial		Lazy binomial	
	worst	worst	amort	worst	amort
INSERT	$\log n$	$\log n$	1	1	1
DECREASE	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$
DELETEMIN	$\log n$	$\log n$	$\log n$	n	$\log n$

Question

Can we develop a heap with faster DELETEMIN than $\mathcal{O}(\log n)$ and INSERT in time $\mathcal{O}(1)$?

Next goal

We need faster operation DECREASE.

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 priority of a given node?

Outline

Fibonacci heap (Fredman, Tarjan, 1987 [5])

Basic properties and legal operations ①

1. Fibonacci heap is a list of trees satisfying the heap invariant ②
2. The order of a tree is the number of children of its root ③
3. We are allowed to join two trees of the same order ④
4. We are allowed to disconnect one child from any node (except the root)
   - The structure of a node contains a bit (mark) to remember whether it has lost a child
5. Roots may loose arbitrary many children
   - If a node become a root, it is unmarked
   - If a root is joined to a tree, it can loose at most one child until it become a root again
6. We are allowed to create new tree with a single element ⑤
7. We are allowed to delete the root of a tree ⑥

Operations which same in lazy binomial heaps

INSERT, FINDMIN, DELETEMIN

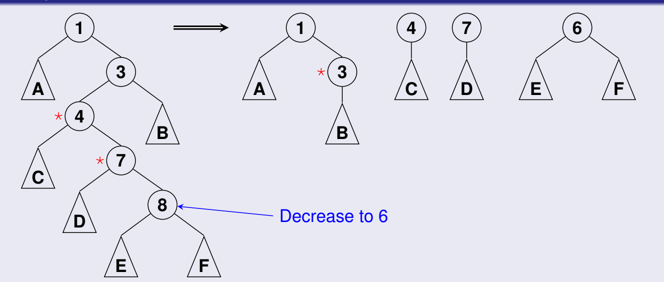
1. Data structures studied so far are defined by structural invariants and operations are designed to keep these invariants. However, Fibonacci heap is defined by elementary operations and the structure is derived from these operations.
2. This is similar as binomial heap but trees are not required to be isomorphic to binomial heaps.
3. This is similar as binomial heap but relations to the number of nodes and height are different.
4. Similarly as in binomial heaps, we append the root of one tree to the list of children of the root of the other tree.
5. Similar as in lazy binomial heaps.
6. Operation DELETEMIN is the same as in lazy binomial heaps including the reconstruction.

Fibonacci heap: Operation DECREASE

Idea

- Decrease priority of a given node and disconnect it from its parent
- If the parent is marked, disconnect it from the grandparent
- If the grandparent is marked, disconnect it
- Repeat until an unmarked node or a root is reached

Example



Fibonacci heap: DECREASE

Algorithm

```
Input: A node u and new priority k
Decrease priority of the node u
if u is a root or the parent of u has priority 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: 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. ①

Proof (The invariant always holds.)

We analyze the initialization and all basic operations allowed by the definition of Fibonacci heap.

1. Initialization: An empty heap satisfies the invariant.
2. INSERT: A tree with a single node satisfies the invariant.
3. Delete a root: Children of remaining nodes are unchanged.
4. Join: A node u of order k is appended as a $(k+1)$-th child of a node p of order k
5. Removing i-th child x from a node u of order k which is a root
6. Removing i-th child x from an unmarked node u of order k which is j-th child of p

1. We assume that later inserted child has a larger index.

Fibonacci heap: Structure

Invariant

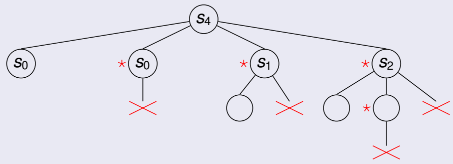
For every node u and its i-th child v holds that v has at least

- $i-2$ 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 $s_k \geq s_{k-2} + s_{k-3} + s_{k-4} + \cdots + s_2 + s_1 + s_0 + 1$.

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

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

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 DELETEMIN.

Fibonacci heap: Complexity

Worst-case complexity

- Operation INSERT: $\mathcal{O}(1)$
- Operation DECREASE: $\mathcal{O}(n)$
- Operation DELETEMIN: $\mathcal{O}(n)$

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

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

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	$\log n$	$\log n$	$\log n$	$\log n$	$\log n$	n	1
DELETEMIN	$\log n$	$\log n$	$\log n$	n	$\log n$	n	$\log n$

Outline

Techniques for memory hierarchy

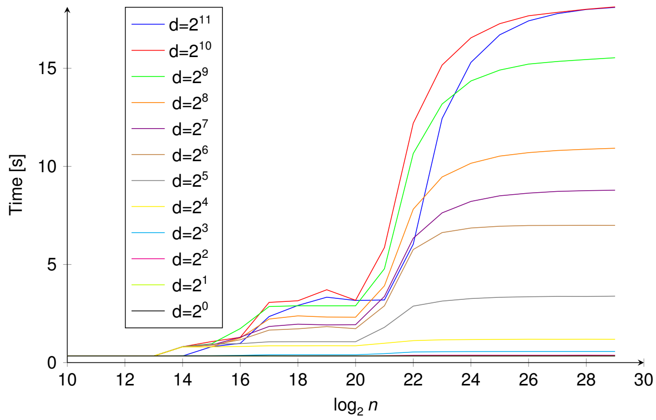
Example of sizes and speeds of different types of memory

	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
SDD	112 GB	448 MB/s
HDD	2 TB	112 MB/s
Internet	∞	10 MB/s

A trivial program

```
# Initialize an array A of 32-bit integers of length n
for (i=0; i+d<n; i+=d) do
    A[i] = i+d # Create a loop using every d-th position
A[i]=0 # Close the loop
for (j=0; j< 2^28; j++) do
    i=A[i] # Repeatedly walk on the loop
```


Memory hierarchy: Trivial program



Memory models

Two level memory model

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. ②
5. The number of page transfers between disk and cache in counted. ③

Cache-oblivious model (Frigo, Leiserson, Prokop, Ramachandran, 1999 [6])

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.

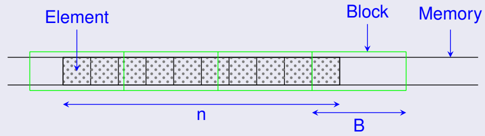
Cache is assumed to be fully associative. ④

1. Also called a block or a line.
2. Whole block must be loaded into cache when data are accessed.
3. For simplicity, we consider only loading pages from disk to cache, which is also called page faults.
4. We assume that every block from disk can be stored in any position of cache. This assumtion makes analysis easier, although it does not hold in practice, see e.g. https://en.wikipedia.org/wiki/CPU_cache#Associativity.

Cache-oblivious analysis: Scanning

Scanning of an array (finding maximum, sum, etc.)

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



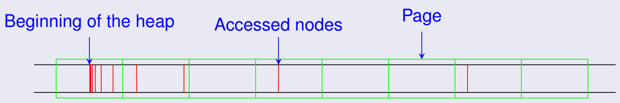
- The number of page transfers is at most $\lceil n/B \rceil + 1$.
- We are allowed to use $\mathcal{O}(1)$ registers to store iterators, sum, etc.

Array reversal

Assuming $P \geq 2$, the number of page transfers is at most $\lceil n/B \rceil + 1$.

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. $\Theta(\log n - \log B) = \Theta(\log \frac{n}{B})$ 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.

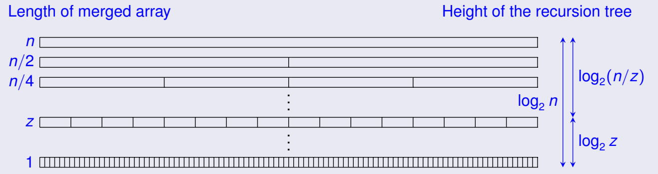
1. One page stores B nodes, so the one page stores a tree of height $\log_2(B) + \mathcal{O}(1)$, if the root is well aligned.
2. More precisely: $\Theta(\max\{1, \log n - \log B\})$

Cache-oblivious analysis: 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: 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

- $B < k$: One page stores at most one row of the matrix.
- $P < k$: 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

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.

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 $\sum_{i=1}^{k-P} i = \Omega\left((k-P)^2\right)$.

Cache-oblivious analysis: Matrix transposition: Cache-aware approach

Cache-aware algoritmus for transposition of a matrix A of size $k \times k$

```
# We split the matrix A into submatrices of size z × z
for (i = 0; i < k; i+ = z) do
    for (j = i; j < k; j+ = z) do
        # We transpose the submatrix starting on position (i, j)
        for (ii = i; ii < min(k, i + z); ii + +) do
            for (jj = max(j, ii + 1); jj < min(k, j + z); jj + +) do
                Swap(A_ii,jj, A_jj,ii)
```

Notes

- Assuming $4B \leq P$, we choose $z = B$
- Every submatrix is stored in at most $2z$ blocks and two submatrices are stored in cache for transposition
- The number of transfered block is at most $\left(\frac{k}{z}\right)^2 2z = \mathcal{O}\left(\frac{k^2}{B}\right)$
- Optimal value of the parameter z depend on computer
- We efficiently use only one level of cache
- This approach is usually faster than cache-oblivious if z is chosen correctly

Cache-oblivious analysis: Matrix transposition: Recursive approach

Idea

Recursively split the matrix into sub-matrices:

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

Matrices A_{11} and A_{22} are transposed using the same scheme, while A_{12} and A_{21} are swapped during their recursive transposition.

Cache-oblivious analysis: Matrix transposition: Recursive approach

```
Procedure transpose_on_diagonal(A)
    if matrix A is small then
        Transpose matrix A using the trivial approach
    else
        A11, A12, A21, A22 ← coordinates of submatrices
        transpose_on_diagonal(A11)
        transpose_on_diagonal(A22)
        transpose_and_swap(A12, A21)

Procedure transpose_and_swap(A, B)
    if matrices A and B are small then
        Swap and transpose matrices A and B using the trivial approach
    else
        A11, A12, A21, A22, B11, B12, B21, B22 ← coordinates of submatrices
        transpose_and_swap(A11, B11)
        transpose_and_swap(A12, B21)
        transpose_and_swap(A21, B12)
        transpose_and_swap(A22, B22)
```


Cache-oblivious analysis: Matrix transposition: Recursive approach

Number of page transfers

1. Tall cache assumption: $M \geq 4B^2$
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. One submatrix $z \times z$ is stored in at most $2z \leq 2B$ blocks
5. Two submatrices $z \times z$ can be stored in cache and their transposition requires at most $4z$ transfers
6. There are $(k/z)^2$ sub-matrices of size $z \times z$
7. The number of transfers is $\mathcal{O}(k^2/B)$
8. This approach is optimal up-to a constant factor

Cache-oblivious analysis: Representation of binary trees

Goals

Construct a representation of binary trees efficiently using cache.
We count the number of transfered blocks during a walk from a leaf to the root.

Representation of a binary heap in an array

Very inefficient: The number of transfered blocks is $\Theta(\log n - \log B) = \Theta(\log \frac{n}{B})$

B-regular heap (similarly B-tree)

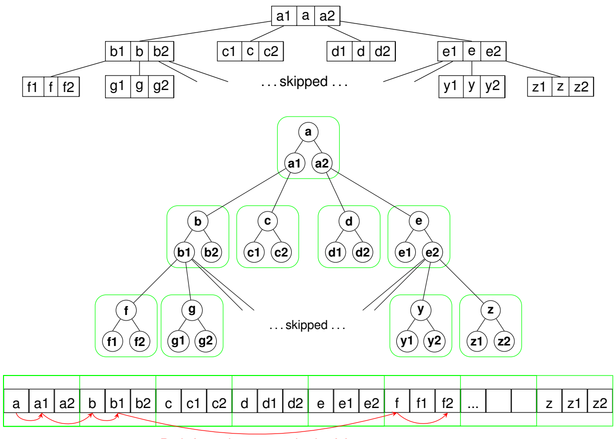
- The height of a tree is $\log_B(n) + \Theta(1)$ ①
- One node is stored in at most two blocks
- The number of transfered blocks is $\Theta(\frac{\log n}{\log B}) = \Theta(\log_B(n))$ ②
- Disadvantage: cache-aware and we want a binary tree

Convert to a binary tree

Every node of a B-regular heap can be replaced by a binary subtree.

1. B-tree has height $\Theta(\log_B(n))$.
2. This is asymptotically optimal — proof is based on Information theory.

Cache-oblivious analysis: Cache-aware representation



Path from the root to the leaf $f2$

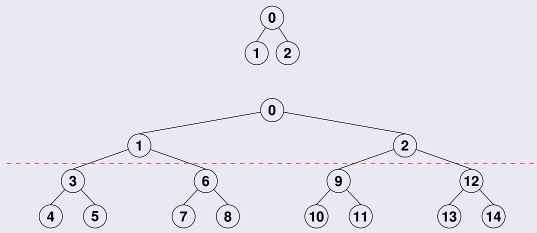
Cache-oblivious analysis: The van Emde Boas layout

Recursive description

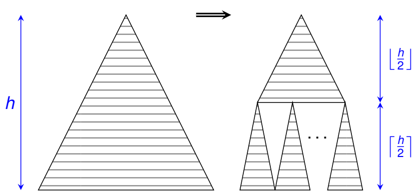
- Van Emde Boas layout of order 0 is a single node.
- The layout of order k has one "top" copy of the layout of order $k-1$ and every leaf of the "top" copy has attached roots of two "bottom" copies of the layout of order $k-1$ as its children.

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: $z \leq \log_2 B \leq 2z$.
- The number of subtrees of height z on the path from the root to a leaf is $\frac{h}{z} \leq \frac{2\log_2 n}{\log_2 B} = 2\log_B n$
- Hence, the number of page transfers is $\mathcal{O}(\log_B n)$.

Cache-oblivious analysis: Comparison of LRU and OPT strategies

Theorem (Sleator, Tarjan, 1985 [18])

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

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

Corollary

If LRU can use twice as many cache pages as OPT, then LRU transports at most twice many pages than OPT does (plus P_{OPT}).

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

Proof ($F_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}} F_{\text{OPT}} + P_{\text{OPT}}$)

1. If LRU transfers $f \leq P_{\text{LRU}}$ blocks in a sequence s, then OPT transfers at least $f - P_{\text{OPT}}$ blocks in s
   - If LRU reads f different blocks in s, then s contains at least f different blocks
   - If LRU reads one block twice in s, then s contains at least $P_{\text{LRU}} \geq f$ different blocks
   - OPT stores at most P_{OPT} blocks of s in cache before processing s, so at least $f - P_{\text{OPT}}$ must be read to cache when s is processed
2. Split sequence $s_1, \ldots, s_k$ into subsequences so that LRU transfers P_{LRU} blocks in every subsequence (except the last one)
3. If F'_{OPT} and F'_{LRU} denotes the number of transfered blocks in a subsequence, then $F'_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}} F'_{\text{OPT}}$ (except the last one)
   - OPT transfers $F'_{\text{OPT}} \geq P_{\text{LRU}} - P_{\text{OPT}}$ blocks in every subsequence
   - Hence, $\frac{F'_{\text{LRU}}}{F'_{\text{OPT}}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}}$
4. The last subsequence satisfies $F''_{\text{LRU}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}} F''_{\text{OPT}} + P_{\text{OPT}}$
   - So, $F''_{\text{OPT}} \geq F''_{\text{LRU}} - P_{\text{OPT}}$ a $1 \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}}$
   - Therefore, $F''_{\text{LRU}} \leq F''_{\text{OPT}} + P_{\text{OPT}} \leq \frac{P_{\text{LRU}}}{P_{\text{LRU}} - P_{\text{OPT}}} F''_{\text{OPT}} + P_{\text{OPT}}$

Comparison of reading and writing data

Reading from memory

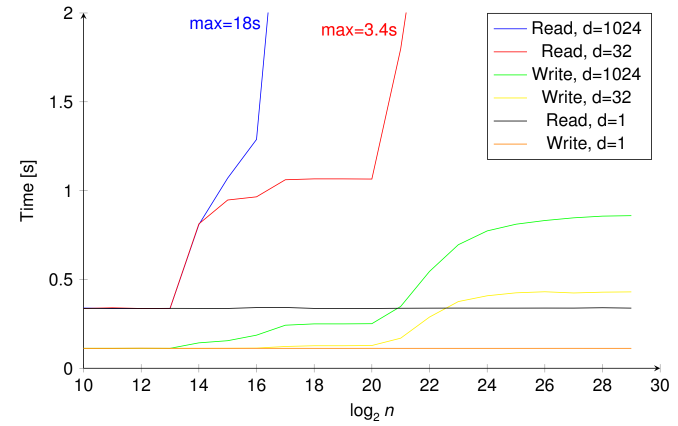
```
# Initialize an array A of 32-bit integers of length n
for (i=0; i+d<n; i+=d) do
    A[i] = i+d # Create a loop using every d-th position
A[i]=0 # Close the loop
for (j=0; j< 2^28; j++) do
    i=A[i] # Repeatedly walk on the loop
```

Writing into memory

```
for (j=0; j< 2^28; j++) do
    A[(j*d) % n] = j # Repeated operation write on d-th position
```
①

1. Actually, we tested A[(j*d) & (n-1)] = j as discussed later.

Comparison of reading and writing data



Few more thicks

Which version is faster and how much?

```
# Modulo
for (j=0; j< 2^28; j++) do
    A[(j*d) % n] = j
# Bitwise conjunction
mask = n − 1 # Assume that n is a power of two
for (j=0; j< 2^28; j++) do
    A[(j*d) & mask] = j
```

How fast is the computation if we skip the last line?

```
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]
printf("%d\n", i);
```


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 all elements of S are hashed to the same position.

Notes

- There is no function "well hashing" every subset $S \subseteq U$.
- For a given subset $S \subseteq U$ we can construct a perfect hashing function.
- We construct a system of hashing functions $\mathcal{H}$ such that for every subset S the expected number of collisions is small for randomly chosen $h \in \mathcal{H}$.

Outline

Universal hashing

Goal

We construct a system $\mathcal{H}$ of hashing functions $f : U \to M$ such that uniformly chosen function $h \in \mathcal{H}$ has for every subset S small expected number of collisions.

Totally random hashing system

- The system $\mathcal{H}$ contains all functions $f : U \to M$
- Hence, $P[h(x) = z] = \frac{1}{m}$ for every $x \in U$ and $z \in M$
- Positions $h(x)$ a $h(y)$ are independent for two different keys $x, y \in U$
- Impractical: encoding one function from $\mathcal{H}$ requires $\Theta(u \log m)$ bits
- We use it in proofs

Hashing random data

- If we need to store a uniformly chosen subset $S \subseteq U$.
- Every reasonable function $f : U \to S$ is sufficient e.g. $f(x) = x \bmod m$.
- Useful in proofs considering totally random hashing system.
- Keys have uniform distributions only in rare practical situations

Universal hashing

c-universal hashing system

A system $\mathcal{H}$ of hashing functions is c-universal, if for every $x, y \in U$ with $x \neq y$ the number of functions $h \in \mathcal{H}$ satisfying $h(x) = h(y)$ is at most $\frac{c|\mathcal{H}|}{m}$ where $c \geq 1$.
Equivalently, a system $\mathcal{H}$ of hashing functions is c-universal, if uniformly chosen $h \in \mathcal{H}$ satisfies $P[h(x) = h(y)] \leq \frac{c}{m}$ for every $x, y \in U$ with $x \neq y$. ①

$(2, c)$-independent hashing system

A set $\mathcal{H}$ of hash functions is $(2, c)$-independent if for every $x_1, x_2 \in U$ with $x_1 \neq x_2$ and $z_1, z_2 \in M$ the number of functions $h \in \mathcal{H}$ satisfying $h(x_1) = z_1$ and $h(x_2) = z_2$ is at most $\frac{c|\mathcal{H}|}{m^2}$. ②
Equivalently, a set $\mathcal{H}$ of hash functions is $(2, c)$-independent if randomly chosen $h \in \mathcal{H}$ satisfies $P[h(x_1) = z_1 \text{ and } h(x_2) = z_2] \leq \frac{c}{m^2}$ for every $x_1 \neq x_2$ elements of U and $z_1, z_2 \in M$.

(k, c)-independent hashing system

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

1. Furthermore, we usually require that $h \in \mathcal{H}$ is computable in $\mathcal{O}(1)$ time and h can be encoded using $\mathcal{O}(1)$ bits.
2. Buckets z_1 and z_2 can be the same but elements x_1 and x_2 must be distinct.

Universal hashing: Relations

Example of c-universal system (Exercise)

$\mathcal{H} = \{h_a(x) = (ax \bmod p) \bmod m;\ 0 < a < p\}$, where $p > u$ is a prime

Relations

1. If a hashing system is (k, c)-independent, then it is $(k-1, c)$-independent. ①
2. If a hashing system is $(2, c)$-independent, then it is also c-universal.
3. 1-independent hashing system may not be c-universal. ②
4. If $P[h(x_i) = z_i \text{ for every } i = 1, \dots, k] \le \frac{1}{m^k}$ for every $z_1, \dots, z_k \in M$, then $P[h(x_i) = z_i \text{ for every } i = 1, \dots, k] = \frac{1}{m^k}$ for every $z_1, \dots, z_k \in M$
5. There exists $z_1, \dots, z_k \in M$ such that $P[h(x_i) = z_i \text{ for every } i = 1, \dots, k] \ge \frac{1}{m^k}$.

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

Universal hashing: Multiply-mod-prime

Definition

- p is a prime greater than u and $[p]$ denotes $\{0, \dots, p-1\}$
- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$ ①
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$
- System $\mathcal{H}$ is 1-universal and 2-independent, ale but it is not 3-independent

Notation

We write $a \equiv_c b$ if $a \bmod c = b \bmod c$ where $a, b \in \mathbb{Z}$ and $c \in \mathbb{N}$.

Lemma

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
Furthermore, these equations define a bijection between $\{(a, b) \in [p]^2;\ a \neq 0\}$ and $\{(y_1, y_2) \in [p]^2;\ y_1 \neq y_2\}$. ③

1. By convention, operator mod has lower priority than addition and multiplication.
2. Subtracting these equations, we get $a(x_1 - x_2) \equiv_p y_1 - y_2$. 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).
3. Indeed, $y_1 = y_2$ if and only if $a = 0$.

Universal hashing: Multiply-mod-prime

Definition

- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$ where p is a prime larger than u
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$

Lemma

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 $\{(a, b) \in [p]^2;\ a \neq 0\}$ and $\{(y_1, y_2) \in [p]^2;\ y_1 \neq y_2\}$.

The multiply-mod-prime set of functions $\mathcal{H}$ is 1-universal

1. For $x_1 \neq x_2$ we have a collision $h_{a,b}(x_1) = h_{a,b}(x_2) \Leftrightarrow y_1 \equiv_m y_2$ and $y_1 \neq y_2$.
2. For given y_1 there are at most $\lceil \frac{p}{m} \rceil - 1$ values $y_2 \neq y_1$ such that $y_1 \equiv_m y_2$.
3. The number such a pairs (y_1, y_2) is at most $p(\lceil \frac{p}{m} \rceil - 1) \le p(\frac{p+m-1}{m} - 1) \le \frac{p(p-1)}{m}$.
4. There are at most $\frac{p(p-1)}{m}$ pairs from $\{(a, b) \in [p]^2;\ a \neq 0\}$ causing a collision $h_{a,b}(x_1) = h_{a,b}(x_2)$.
5. Hence, $P[h_{a,b}(x_1) = h_{a,b}(x_2)] \le \frac{p(p-1)}{m|\mathcal{H}|} \le \frac{1}{m}$.

Universal hashing: Multiply-mod-prime

Definition

- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$ where p is a prime larger than u
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$

Lemma

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 $(y_1, y_2) \in [p]^2$.

The multiply-mod-prime set of functions $\mathcal{H}$ is 2-independent

1. The number of y_1 with $z_1 = y_1 \bmod m$ is at most $\lceil \frac{p}{m} \rceil$.
2. The number of (y_1, y_2) with $z_1 = y_1 \bmod m$ and $z_2 = y_2 \bmod m$ is at most $\lceil \frac{p}{m} \rceil^2$.
3. The number of (x_1, x_2) with $h_{a,b}(x_1) = z_1$ and $h_{a,b}(x_2) = z_2$ is at most $\lceil \frac{p}{m} \rceil^2$.
4. $P[h_{a,b}(x_1) = z_1 \text{ and } h_{a,b}(x_2) = z_2] \le \lceil \frac{p}{m} \rceil^2 \frac{1}{p(p-1)} \le (\frac{p+m}{m})^2 \frac{2}{p^2} \le \left(\frac{2p}{m}\right)^2 \frac{2}{p^2} = \frac{8}{m^2}$

Universal hashing: Multiply-mod-prime

Definition

- $h_{a,b}(x) = (ax + b \bmod p) \bmod m$ where p is a prime larger than u
- $\mathcal{H} = \{h_{a,b};\ a, b \in [p], a \neq 0\}$

System $\mathcal{H}$ is not 3-independent

1. For simplicity, assume that $p \ge 4u$.
2. Choose arbitrary $z_1, z_2 \in [p]$.
3. Let $x_1 = 1$, $x_2 = 3$, $x_3 = 2$ and $z_3 = (z_1 + z_2)2^{-1}$ where $2^{-1} \in GF(p)$.
4. Note that $3a + b < p$ for every $a, b \in [p]$.
5. If $h(x_1) = z_1$ and $h(x_2) = z_2$, then $h(x_3) = z_3$.
   - $2h(x_3) \equiv_p 2(2a + b) = (a + b) + (3a + b) \equiv_p z_1 + z_2 \equiv_p 2z_3$
6. Conditional probability: $P[h(x_3) = z_3 | h(x_1) = z_1 \text{ and } h(x_2) = z_2] = 1$
7. $P[h(x_3) = z_3 \text{ and } h(x_1) = z_1 \text{ and } h(x_2) = z_2]$
$= P[h(x_3) = z_3 | h(x_1) = z_1 \text{ and } h(x_2) = z_2]\ P[h(x_1) = z_1 \text{ and } h(x_2) = z_2]$
$= P[h(x_1) = z_1 \text{ and } h(x_2) = z_2] \ge \frac{1}{m^2}$ for some $z_1, z_2 \in [p]$
8. Hence, for every $c \ge 1$ there exists $m \in \mathbb{N}$ and $p \ge 4u \ge 4m$ and $z_1, z_2, z_3 \in [p]$ such that $P[h(x_1) = z_1 \text{ and } h(x_2) = z_2 \text{ and } h(x_3) = z_3] > \frac{c}{m^3}$.

Universal hashing: Poly-mod-prime

Hashing system Poly-mod-prime

- Let p be a prime larger than u and $k \geq 2$ be an interger
- $h_{a_0,\dots,a_{k-1}}(x) = (\sum_{i=0}^{k-1} a_i x^i \bmod p) \bmod m$
- $\mathcal{H} = \{h_{a_0,\dots,a_{k-1}};\ a_0,\dots,a_{k-1} \in [p]\}$
- Note that Poly-mod-prime for $k = 2$ is the same as Mupliply-mod-prime

k-independence (exercise)

Hashing system Poly-mod-prime is k-independent but it is not $(k+1)$-independent

Universal hashing: Multiply-shift

Multiply-shift

- Assume that $u = 2^w$ a $m = 2^l$ where w and l are integers
- $h_a(x) = (ax \bmod 2^w) >> (w - l)$
- $\mathcal{H} = \{h_a;\ a$ odd w-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)

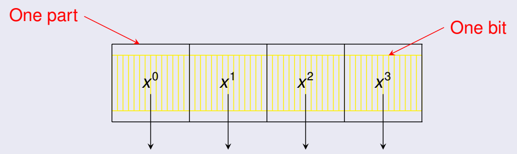
Hashing system Multiply-shift is 2-independent.

Universal hashing: Tabular hashing

Tabular hashing

- Assume that $u = 2^w$ and $m = 2^l$ and w is a multiple of an integer d
- Binary code of $x \in U$ is split to d parts $x^0, \dots, x^{d-1}$ by $\frac{w}{d}$ bits
- For every $i \in [d]$ generate a totally random hashing function $T_i : [2^{w/d}] \to M$
- Hashing function is $h(x) = T_0(x^0) \oplus \dots \oplus T_{d-1}(x^{d-1})$ ①

Example



$h(x) = T_0(x^0) \oplus T_1(x^1) \oplus T_2(x^2) \oplus T_3(x^3)$

Universality

Tabular hashing is 3-independent, but it is not 4-independent.

1. $\oplus$ denotes bit-wise exclusive or (XOR).

Universal hashing: Tabular hashing

Tabular hashing

- Assume that $u = 2^w$ and $m = 2^l$ and w is a multiple of an integer d
- Binary code of $x \in U$ is split to d parts $x^0, \dots, x^{d-1}$ by $\frac{w}{d}$ bits
- For every $i \in [d]$ generate a totally random hashing function $T_i : [2^{w/d}] \to M$
- Hashing function is $h(x) = T_0(x^0) \oplus \dots \oplus T_{d-1}(x^{d-1})$

Universality

Tabular hashing is 3-independent, but it is not 4-independent.

Proof of 2-independence

- Consider two elements x_1 a x_2 which differ in i-th parts
- Let $h_i(x) = T_0(x^0) \oplus \dots \oplus T_{i-1}(x^{i-1}) \oplus T_{i+1}(x^{i+1}) \oplus \dots \oplus T_{d-1}(x^{d-1})$
- $P[h(x_1) = z_1] = P[h_i(x_1) \oplus T_i(x_1^i) = z_1] = P[T_i(x_1^i) = z_1 \oplus h_i(x_1)] = \frac{1}{m}$ ①
- Random events $h(x_1) = z_1$ and $h(x_2) = z_2$ are independent
  - Random variables $T_i(x_1^i)$ and $T_i(x_2^i)$ are independent
  - Random events $T_i(x_1^i) = z_1 \oplus h_i(x_1)$ and $T_i(x_2^i) = z_2 \oplus h_i(x_2)$ are independent
- $P[h(x_1) = z_1$ and $h(x_2) = z_2] = P[h(x_1) = z_1]P[h(x_2) = z_2] = \frac{1}{m^2}$

1. $T_i(x_1^i)$ has the uniform distribution on M and random variables $T_i(x_1^i)$ and $z_1 \oplus h_i(x_1)$ are independent.

Universal hashing: Tabular hashing

Tabular hashing is not 4-independent

1. Consider elements x_1, x_2, x_3 and x_4 such that
   - x_1 satisfies $x_1^0 = 0$, $x_1^1 = 0$, $x_1^i = 0$ for $i \geq 2$
   - x_2 satisfies $x_2^0 = 1$, $x_2^1 = 0$, $x_2^i = 0$ for $i \geq 2$
   - x_3 satisfies $x_3^0 = 0$, $x_3^1 = 1$, $x_3^i = 0$ for $i \geq 2$
   - x_4 satisfies $x_4^0 = 1$, $x_4^1 = 1$, $x_4^i = 0$ for $i \geq 2$
2. Observe that $h(x_4) = h(x_1) \oplus h(x_2) \oplus h(x_3)$
3. Choose arbitrary $z_1, z_2, z_3 \in M$ and let $z_4 = z_1 \oplus z_2 \oplus z_3$
4. Conditional probability
   $P[h(x_4) = z_4 | h(x_1) = z_1$ and $h(x_2) = z_2$ and $h(x_3) = z_3] = 1$
5. $P[h(x_4) = z_4$ and $h(x_1) = z_1$ and $h(x_2) = z_2$ and $h(x_3) = z_3] =$
   $P[h(x_1) = z_1$ and $h(x_2) = z_2$ and $h(x_3) = z_3] \geq \frac{1}{m^3}$ for some $z_1, z_2, z_3 \in M$
6. Hence, for every $c \geq 1$ there exists $m \in \mathbb{N}$ and $z_1, z_2, z_3, z_4 \in M$ such that
   $P[h(x_1) = z_1$ and $h(x_2) = z_2$ and $h(x_3) = z_3$ and $h(x_4) = z_4] > \frac{c}{m^4}$.

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 $M = [2^l]$ and let $v \geq w + l - 1$
- $h_{a_1,\dots,a_d,b}(x_1, \dots, x_d) = ((b + \sum_{i=1}^{d} a_i x_i) >> (l - v)) \bmod m$
- $\mathcal{H} = \{h_{a_1,\dots,a_d,b};\ a_1, \dots, a_d, b \in [2^v]\}$
- $\mathcal{H}$ is 2-universal

Multiply-mod-prime 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-mod-prime for variable-length string II

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

Outline

Separate chaining

Description

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

- linked list
- dynamic array
- self-balancing tree

Implementations

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

Separate chaining: Example

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

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

Terminology

We say that a hashing system $\mathcal{H}$ is strongly k-independent, if H is $(k,1)$-independent. Note that H is $(k,1)$-independent if randomly chosen $h \in \mathcal{H}$ satisfies $P[h(x_i) = z_i$ for every $i = 1, \dots, k] = \frac{1}{m^k}$ for every pair-wise different elements $x_1, \dots, x_k \in U$ and $z_1, \dots, z_k \in M$.

Separate chaining: Analysis

Definition

- $\alpha = \frac{n}{m}$ is the load factor; we assume that $\alpha = \Theta(1)$
- 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

Expected chain length (number of elements in a bucket)

If hashing system is strongly 1-independent, the number of elements in one bucket $j \in M$ is $E[A_j] = \alpha$. ①

Observations

If hashing system is strongly 2-independent, then

1. $E[A_j^2] = \alpha(1 + \alpha - 1/m)$ ②
2. $Var(A_j) = \alpha(1 - 1/m)$ ③

1. $E[A_j] = E[\sum_{i \in S} I_{ij}] = \sum_{i \in S} E[I_{ij}] = \sum_{i \in S} P[h(i) = j] = \sum_{i \in S} \frac{1}{m} = \frac{n}{m}$ where the second equality follows from the linearity of expectation, the third one from the definition of expectation and the last one from 1-independence.
2. $E[A_j^2] = E[(\sum_{i \in S} I_{ij})(\sum_{k \in S} I_{kj})] = \sum_{i \in S} E[I_{ij}^2] + \sum_{i,k \in S, i \neq k} E[I_{ij} I_{kj}] =$
$= \sum_{i \in S} P[h(i) = j] + \sum_{i,k \in S, i \neq k} E[h(i) = j \text{ and } h(k) = j] = \alpha + \frac{n(n-1)}{m^2}$
3. $Var(A_j) = E[A_j^2] - E^2[A_j] = \alpha(1 + \alpha - 1/m) - \alpha^2$

Separate chaining: Operation Find

Expected number of comparisons during a successful operation Find

- The total number of comparisons to find all elements is divided by the number of elements
- Strongly 2-independent hashing system is considered
- The total number of comparisons is $\sum_j \sum_{k=1}^{A_j} k = \sum_j \frac{A_j(A_j+1)}{2}$
- Expected number of comparisons is $1 + \frac{\alpha}{2} - \frac{1}{2m}$
  - $E\left[\frac{1}{n} \sum_j \frac{A_j(A_j+1)}{2}\right] = \frac{1}{2n}(E[\sum_j A_j] + \sum_j E[A_j^2]) = \frac{1}{2n}(n + m\alpha(1 + \alpha - \frac{1}{m}))$

Expected number of comparisons during an unsuccessful operation Find

- The number of comparisons during an unsuccessful Findof element x is the number of elements $i \in S$ satisfying $h(i) = h(x)$
- We need to count $E[|\{i \in S;\ h(i) = h(x)\}|]$
- c-universal hashing system is considered
- Expected number of comparisons is $c\alpha$
  - $E[|\{i \in S;\ h(i) = h(x)\}|] = \sum_{i \in S} P[h(i) = h(x)] \leq \sum_{i \in S} \frac{c}{m} = c\alpha$

Separate chaining: The longest chain

Definition

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

Length of the longest chain

If $\alpha = \Theta(1)$ and the hashing system is totally independent, then the length of the longest chain is $\max_{j \in M} A_j = \Theta(\frac{\log n}{\log \log n})$ with high probability. This also holds for

- $\frac{\log n}{\log \log n}$-independent hashing system (Schmidt, Siegel, Srinivasan, 1995 [17])
- tabular hashing (Pătraşcu, Thorup, 2012 [15])

Expected length of the longest chain (Exercise)

If $\alpha = \Theta(1)$ and the hashing system is totally independent, then the expected length of the longest chain is $\Theta(\frac{\log n}{\log \log n})$ elements.

Chernoff Bound

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

Separate chaining: The longest chain

Length of the longest chain

If $\alpha = \Theta(1)$ and the hashing system is totally independent, then the length of the longest chain is $\max_{j \in M} A_j = \Theta(\frac{\log n}{\log\log n})$ with high probability.

Chernoff Bound

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

Proof: $\max_{j \in M} A_j = \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ with high probability

- Consider $\epsilon > 0$ a $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$. So, $c\mu = (1+\epsilon)\frac{\log n}{\log\log n}$
- It holds $P[\max_j A_j > c\mu] = P[\exists j : A_j > c\mu] \le \sum_j P[A_j > c\mu] = mP[A_1 > c\mu]$
- Using the Chernoff bound on I_{i1} for $i \in S$: $\mu = E[A_1] = \alpha$
- It holds $P[\max_j A_j > c\mu] \le mP[A_1 > c\mu] < me^{-\mu}e^{c\mu - c\mu\log c}$

Separate chaining: The longest chain

Proof: $\max_{j \in M} A_j = \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ with high probability

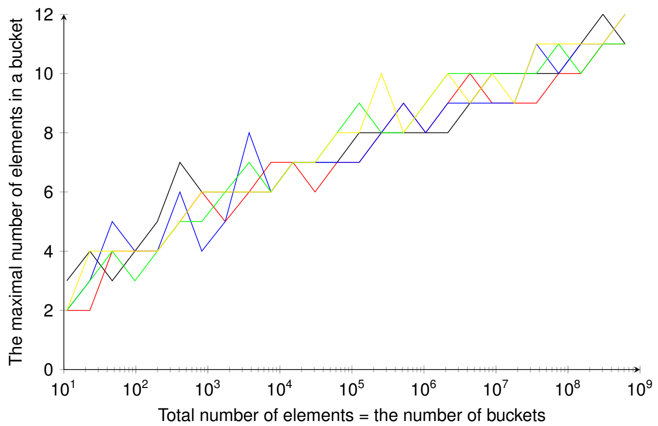
- Consider $\epsilon > 0$ a $c = (1+\epsilon)\frac{\log n}{\mu \log\log n}$. So, $c\mu = (1+\epsilon)\frac{\log n}{\log\log n}$

$$
\begin{aligned}
P[\max_j A_j > c\mu] &< me^{-\mu}e^{c\mu - c\mu\log c} \\
&= me^{-\mu}e^{(1+\epsilon)\frac{\log n}{\log\log n} - (1+\epsilon)\frac{\log n}{\log\log n}\log\left(\frac{(1+\epsilon)\log n}{\mu\log\log n}\right)} \\
&= me^{-\mu}e^{(1+\epsilon)\frac{\log n}{\log\log n} - (1+\epsilon)\log n + (1+\epsilon)\frac{\log n}{\log\log n}\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)} \\
&= me^{-\mu}e^{(1+\epsilon)\frac{\log n}{\log\log n} - (1+\epsilon)\log n + (1+\epsilon)\frac{\log n}{\log\log n}\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)} \\
&= me^{-\mu}n^{\frac{1+\epsilon}{\log\log n} - (1+\epsilon) + \frac{1+\epsilon}{\log\log n}\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)} \\
&= \frac{m}{n^{1+\frac{\epsilon}{2}}}e^{-\mu}n^{-\frac{\epsilon}{2} + \frac{1+\epsilon}{\log\log n} + (1+\epsilon)\frac{\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)}{\log\log n}} \\
&< \frac{1}{\alpha n^{\frac{\epsilon}{2}}}e^{-\mu}n^0 < \frac{1}{n^{\frac{\epsilon}{3}}} \qquad \text{...for sufficiently large } n
\end{aligned}
$$

Since $-\frac{\epsilon}{2} + \frac{1+\epsilon}{\log\log n} + (1+\epsilon)\frac{\log\left(\frac{\mu}{1+\epsilon}\log\log n\right)}{\log\log n} < 0$ for sufficiently large n.

- Hence, $P[\max_j A_j \le (1+\epsilon)\frac{\log n}{\log\log n}] > 1 - \frac{1}{n^{\frac{\epsilon}{3}}}$.

Separate chaining: Length of the longest chain (5 experiments)



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

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

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

Outline

Linear probing

Goal

Store elements directly in the table to reduce overhead.

Linear probing: Operation INSERT

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

Operation FIND

Iterate until the given key or empty bucket is fount.

Operation DELETE

- A lazy version: Flag the bucket of deleted element to ensure that the operation Find continues searching.
- A version without flags: Check and move elements in a chain (Exercise)

Linear probing: Complexity

Assumptions

- $m \ge (1+\epsilon)n$ for some $\epsilon > 0$
- No operation DELETE

Expected number of comparisons during operation INSERT

- $\mathcal{O}(\frac{1}{\epsilon^2})$ for totally random hashing systems (Knuth, 1963 [10])
- constant for $\log(n)$-independend hashing systems (Schmidt, Siegel, 1990 [16])
- $\mathcal{O}\left(\frac{1}{\epsilon^{\frac{13}{6}}}\right)$ for 5-independent hashing systems (Pagh, Pagh, Ruzic, 2007 [12])
- $\Omega(\log n)$ for some 4-independent hashing system (Pătraşcu, Thorup, 2010 [14]) 1
- $\mathcal{O}(\frac{1}{\epsilon^2})$ for tabular hashing system (Pătraşcu, Thorup, 2012 [15])

1. There exists a 4-independent hashing system and a sequence of operations INSERT such that for a randomly chosen hashing function from the system the expected complexity of is $\Omega(\log n)$.

Linear probing: Analysis

The number of elements from a given bucket to closest empty one

If $\alpha < 1$ and a hashing system is totally independent, then the expected number of key comparisons during operation INSERT is $\mathcal{O}(1)$.

Proof

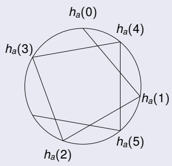
1. Let $1 < c < \frac{1}{\alpha}$ and $q = \left(\frac{e^{c-1}}{c^c}\right)^\alpha$
   - Observe that $0 < q < 1$
2. Let $p_t = P[|\{x \in S;\ h(x) \in T\}| = t]$ be the probability that t elements are hashed into a given set of T buckets. Then, $p_t < q^t$. ①
   - Let X_i be the random variable indication whether an element i is hashed into T
   - Let $X = \sum_{i \in S} X_i$ and $\mu = E[X] = t\alpha$
   - Observe that $c\mu = c\alpha t < t$
   - Chernoff: $p_t = P[X = t] \leq P[X > c\mu] < \left(\frac{e^{c-1}}{c^c}\right)^\mu = q^{\frac{\mu}{\alpha}} = q^t$
3. Let b be a bucket. Let p'_k be the probability that buckets b to $b+k-1$ are occupied and $b+k$ is the first empty bucket. Then $p'_k < \frac{q^k}{1-q}$. ②
   - $p'_k < \sum_{s=0}^\infty p_{s+k} < q^k \sum_{s=0}^\infty q^s = \frac{q^k}{1-q}$
4. The expected number of key comparisons is $\sum_{k=0}^m kp'_k < \frac{1}{1-q}\sum_{k=0}^\infty kq^k = \frac{2-q}{(1-q)^3}$

1. Here, we consider elements that are mapped into given buckets by a hashing function (not inserted by linear probing).
2. Hence, buckets $b-s$ to $b+k-1$ are occupied for some s and $b-s-1$ and $b+k-1$ are empty buckets. All indices of buckets are counted modulus m.

Linear probing: Why 2-independent hashing system is insufficient?

Combination of multiply-shift and linear probing

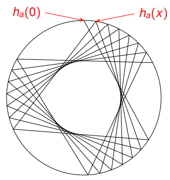
- Multiply-shift: $h_a(x) = (ax \bmod 2^w) >> (w-l)$
- Denote $h'_a(x) = \frac{ax \bmod 2^w}{2^{w-l}} = \frac{ax}{2^{w-l}} \bmod 2^l$
- Then $h_a(x) = \lfloor h'_a(x) \rfloor$
- What is the complexity of inserting elements $S = \{1, \dots, n\}$?



Properties

1. Denote $||x|| = \min\{x, 2^l - x\}$, i.e. the distance between 0 and x along the cycle
2. Then, $||h'_a(x) - h'_a(y)|| = ||h'_a(x-1) - h'_a(y-1)|| = ||h'_a(x-y)||$ for $x \geq y$
3. Hence, $||h'_a(ix)|| \leq i||h'_a(x)||$ for every $i \in \{1, \dots, n\}$
4. If $||h'_a(x)|| \geq 1$ for every $x \in S$, then h_a is perfect on S
5. $P[h_a$ is not perfect$] \leq P[\exists x \in S : ||h'_a(x)|| < 1] \leq \sum_{x \in S} P[||h'_a(x)|| < 1] \leq \frac{2n}{m}$
6. $P[h_a$ is perfect on $S] = 1 - P[h_a$ is not perfect on $S] \geq 1 - \frac{2n}{m}$

Linear probing: Why 2-independent hashing system is insufficient?



Combination of Multiply-shift and linear probing

- Consider element $x \in S$ such that $||h'_a(x)|| \leq \frac{1}{2}$
- Elements $x, 2x, \dots, kx$ belong into $0, 1, \dots, \lfloor \frac{1}{2}k \rfloor$, where $k = \lfloor \frac{n}{x} \rfloor$
- We have at most x groups, each having at least k elements
- Complexity of operation INSERT is $\Omega(\frac{n}{x})$, if $||h'_a(x)|| \leq \frac{1}{2}$

Linear probing and hashing systems (Pătraşcu, Thorup, 2010 [14])

- FIND using Multiply-shift has expected complexity $\Theta(\log n)$
- There exists 2-independent hashing system such that FIND has complexity $\Theta(\sqrt{n})$

Outline

Cuckoo hashing (Pagh, Rodler, 2004 [13])

Description

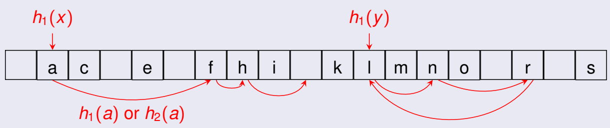
Given two hash functions h_1 and h_2, a key x can be stored in $h_1(x)$ or $h_2(x)$. One position can store at most one element.

Operations FIND and DELETE

Trivial, complexity is $\mathcal{O}(1)$ in worst case.

INSERT: Example

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



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

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

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 of length k is at most $\frac{1}{mc^k}$.

Complexity of operation Insert without rehashing

Let $c > 1$ and $m \geq 2cn$. Expected number of swaps during operation INSERT is $\mathcal{O}(1)$.

Number of rehashes to build Cuckoo hash table

Let $c > 2$ and $m \geq 2cn$. Expected number of rehashes to insert n elements is $\mathcal{O}(1)$.

Proof of the lemma by induction on k:

$k = 1$ For one element x, the probability that x creates an edges between vertices i and j is $P[\{i,j\} = \{h(x), h(y)\}] = \frac{2}{m^2}$. 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:

- The probability that there exists a path from $i = h_1(x)$ to some position j of length k is at most $m\frac{1}{mc^k} = \frac{1}{c^k}$.
- The expected length of the path starting at $h_1(x)$ is at most $\sum_{k=1}^{n} k\frac{1}{c^k} \leq \sum_{k=1}^{\infty} \frac{k}{c^k} = \frac{c}{(c-1)^2}$.

Number of rehashes:

- If rehashes is needed, then the Cuckoo contains a cycle.
- The probability that the graph contains a cycle is at most the probability that there exists two positions i and j joined by an edge and there exists a path between i and j of length $k \geq 2$. So, the Cuckoo graph contains a cycle with probability at most $\binom{m}{2}\sum_{k=2}^{\infty}\frac{1}{mc}\frac{1}{mc^k} \leq \sum_{k=1}^{\infty}\frac{1}{c^k} = \frac{1}{c-1}$.
- The probability that z rehashes are needed to build the table is at most $\frac{1}{(c-1)^z}$.
- The expected number of rehashes is at most $\sum_{z=0}^{\infty} z\frac{1}{(c-1)^z} = \frac{c-1}{(c-2)^2}$.

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 $\mathcal{O}(1)$.

Pagh, Rodler [13]

If $c > 1$ and $m \geq 2cn$ and hashing system is $\log n$-independent then expected amortized complexity of INSERT is $\mathcal{O}(1)$.

Pătraşcu, Thorup [15]

If $c > 1$ and $m \geq 2cn$ and tabular hashing is used, then time complexity to build a static Cuckoo table is $\mathcal{O}(n)$ with high probability.

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.

Outline

1. Amortized analysis
2. Splay tree
3. (a,b)-tree and red-black tree
4. Heaps
5. Cache-oblivious algorithms
6. Hash tables
7. Geometrical data structures
   - Range trees
   - k-d trees
8. Bibliography

Range query

Problem description

- Given set S of n points in $\mathbb{R}^d$
- A range means a d-dimensional rectangle
- Operation QUERY: Find all points of S in a given range
- Operation COUNT: Determine the number of points of S in a given range

Applications

- Computer graphics, computational geometry
- Database queries, e.g. list all employees in age 20–35 and salary 20-30 thousands

Outline

1. Amortized analysis
2. Splay tree
3. (a,b)-tree and red-black tree
4. Heaps
5. Cache-oblivious algorithms
6. Hash tables
7. Geometrical data structures
   - Range trees
   - k-d trees
8. Bibliography

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

Static version

Store all points in an array in the increasing order

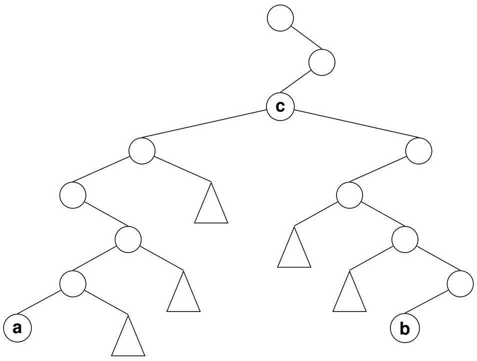
- BUILD: $\mathcal{O}(n \log n)$
- COUNT: $\mathcal{O}(\log n)$
- QUERY: $\mathcal{O}(k + \log n)$ where k is the number of points in the range

Dynamic version

Store all points in a balanced search tree

- BUILD: $\mathcal{O}(n \log n)$
- INSERT: $\mathcal{O}(\log n)$
- DELETE: $\mathcal{O}(\log n)$
- COUNT: $\mathcal{O}(\log n)$
- QUERY: $\mathcal{O}(k + \log n)$

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



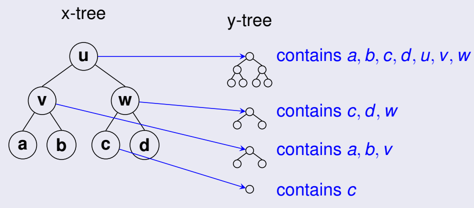
Let a and b be the smallest and the largest elements of S in the range, resp., and c be the deepest common predecessor of a and b.

Range trees in $\mathbb{R}^2$

Construction

- Build a binary search tree according to the x coordinate (called x-tree).
- Let S_u be the set of all points of S in the subtree of a node u.
- For every node u of x-tree build a binary search tree according to y-coordinate containing points of S_u (called y-tree).

Example



Etc.

Range trees in $\mathbb{R}^2$: Space complexity

Vertical point of view

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

Horizontal point of view

Every level of x-tree decomposes S by x-coordinates. Therefore, y-trees corresponding to one level of x-tree contain every point at most once.

Space complexity

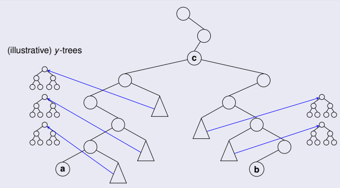
Since every point is stored in $\mathcal{O}(\log n)$ y-trees, the space complexity is $\mathcal{O}(n \log n)$.

Range trees in $\mathbb{R}^2$: QUERY $\langle a_x, b_x \rangle \times \langle a_y, b_y \rangle$

Range query

1. Search for keys a_x and b_x in the x-tree.
2. Identify nodes in the x-tree storing 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.

Range trees in $\mathbb{R}^2$: 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 of a set of points S':

1. Let p be the median of S' by x-coordinate.
2. Create a node u for p in x-tree.
3. Create y-tree of points S' assigned to u. ①
4. Split both sorted arrays by x-coordinate of p.
5. Recursively create both children of u.

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 $\mathcal{O}(|S'|)$.
2. Use master theorem, or observe that building one level of x-tree takes $\mathcal{O}(n)$-time.

Interval trees in $\mathbb{R}^d$ (assuming $d \geq 2$)

Description

- i-tree is a binary search tree by i-th coordinate for $i = 1, \ldots, d$
- For $i < d$ every node u of i-tree has a pointer to the $(i+1)$-tree containing points of S_u
- Range tree means a system of all i-trees for all $i = 1, \ldots, d$

Representation

Structure for a node in range tree stores

Element:	point stored in the node.
Left, Right	pointers to the left and the right child
Tree	pointer to the root of assigned $(i+1)$-tree

Note

Let u be a node of i-tree for some $i = 1, \ldots, d$. Let T be the set of all nodes reachable from u by a sequence of pointers Left, Right and Tree. Then, T forms a range tree of points S_u in coordinates $i, \ldots, d$.

Range tree in $\mathbb{R}^d$: Structure

Observations

- Number of i-trees is at most number nodes in $(i-1)$-trees for $i = 2, \ldots, d$.
- If an $(i-1)$-tree T contains a point p, then p is contained in $\mathcal{O}(\log n)$ i-trees assigned to nodes of T for $i = 2, \ldots, d$.

Number of trees and nodes assuming that trees are balanced

	1-tree	2-tree	d-tree
Number of trees containing a given point	1	$\mathcal{O}(\log n)$	$\mathcal{O}\left(\log^{d-1} n\right)$
Number of trees	1	n	$\mathcal{O}\left(n \log^{d-2} n\right)$
Number of nodes	n	$\mathcal{O}(n \log n)$	$\mathcal{O}\left(n \log^{d-1} n\right)$

Range tree in $\mathbb{R}^d$: Build

Algorithm (Points S are stored in an array sorted by the last coordinate)

Procedure Build (*Set of points S, current coordinate i*)
  if *S is empty or $i > d$* **then**
    return NIL
  $v \leftarrow$ new node
  $v.element \leftarrow$ median of S by i-th coordinate
  $S_l, S_r \leftarrow$ points of S having i-th coordinate smaller (larger) than $v.element$
  $v.tree \leftarrow$ Build $(S, i+1)$
  $v.left \leftarrow$ Build (S_l, i)
  $v.right \leftarrow$ Build (S_r, i)
  return v

Complexity of a single call of Build (without recursion)

- If $i < d$ then complexity is $\mathcal{O}(|S|)$.
- If $i = d$ then complexity is $\mathcal{O}(1)$.

Range tree in $\mathbb{R}^d$: Complexity of Build

Building all d-trees

- Number of nodes in all d-trees is $\mathcal{O}\left(n \log^{d-1} n\right)$
- Complexity of creations of all d-trees is $\mathcal{O}\left(n \log^{d-1} n\right)$

Building all i-trees for some $i = 1, \dots, d-1$ (excluding $(i+1)$-trees

- Number of nodes in all i-trees is $\mathcal{O}\left(n \log^{i-1} n\right)$
- Time complexity of one i-tree T on n_T nodes in $\mathcal{O}(n_T \log n_T)$
- Time complexity of all i-trees is (up-to multiplicative constant)

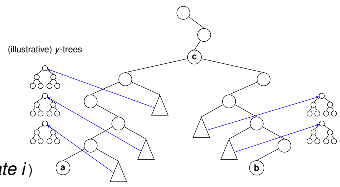
$$\sum_{i\text{-tree } T} n_T \log n_T \le \log n \sum_{i\text{-tree } T} n_T = \log n \cdot n \log^{i-1} n = n \log^i n$$

Time complexity of operation Build

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

Range tree in $\mathbb{R}^d$: Query $\langle a_1, b_1\rangle \times \cdots \times \langle a_d, b_d\rangle$

Procedure Query (*node v, current coordinate i*)
  if $v = NIL$ **then**
    return
  if $v.key \le a_i$ **then**
    Query $(v.right, i)$
  else if $v.key \ge b_i$ **then**
    Query $(v.left, i)$
  else
    Query_left $(v.left, i)$
    Query_right $(v.right, i)$
Procedure Query_left (*node v, coordinate i*)
  if $v = NIL$ **then**
    return
  if $v.key < a_i$ **then**
    Query_left $(v.right, i)$
  else
    Query_left $(v.left, i)$
    if $i < d$ **then**
      Query $(v.right.tree, i+1)$
    else
      Print all points in the subtree of $v.right$



Range tree in $\mathbb{R}^d$: Complexity of Count and Query

Complexity of operation Count

- In every tree, at most $\mathcal{O}(\log n)$ nodes are accessed
- From every visited i-tree at most $\mathcal{O}(\log n)$ assigned $(i+1)$-trees are visited
- The number of visited i-trees is $\mathcal{O}\left(\log^{i-1} n\right)$
- Complexity of operation Count is $\mathcal{O}\left(\log^d n\right)$

Complexity of operation Query

- Complexity printing all points is $\mathcal{O}(k)$, where k is the number of listed points
- Complexity of operation Query is $\mathcal{O}\left(k + \log^d n\right)$

Range trees in $\mathbb{R}^d$: Dynamization using BB[α]-trees

BB[α]-tree

- Binary search tree
- Let s_u be the number of nodes in the subtree of a node u
- Subtrees of both children of u contains at most αs_u nodes

Operation Insert (Delete is analogous)

- Insert/Delete given node similarly as in (non-balanced) binary search trees
- When a node u violates the weight condition, rebuild whole subtree in time $\mathcal{O}(s_u)$.

Amortized cost of rebalancing

- Between two consecutive rebuilds of a node u, there are at least $\Omega(s_u)$ updates in the subtree of u
- Therefore, amortized cost of rebuilding a subtree is $\mathcal{O}(1)$
- 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 $\mathcal{O}(\log n)$.

Range trees in $\mathbb{R}^d$: Dynamization using BB[α]-trees

Using BB[α]-trees in range trees

- Every tree in range tree is a BB[α]-tree
- If operation Insert/Delete requires in BB[α]-tree requires balancing, then all assigned trees are rebuilded.

Amortized complexity of operations Insert and Delete

- Without balancing, there are $\mathcal{O}(\log n)$ accessed nodes in $\mathcal{O}\left(\log^{i-1} n\right)$ accessed i-trees
- Complexity without balancing is $\mathcal{O}\left(\log^d n\right)$; next, we analyze balancing
- Rebuild of a node u in i-tree takes $\mathcal{O}\left(s_u \log^{d-i} s_u\right)$
- Rebuild of u occurs after $\Omega(s_u)$ operations Insert/Delete in the subtree of u
- Amortized cost of operations Insert/Delete is $\mathcal{O}\left(\log^{d-i} s_u\right) \le \mathcal{O}\left(\log^{d-i} n\right)$
- Amortized time is $\sum_{i=1}^{d} \mathcal{O}\left(\log^{i-1} n\right)\mathcal{O}(\log n)\mathcal{O}\left(\log^{d-i} n\right) = \mathcal{O}\left(\log^d n\right)$

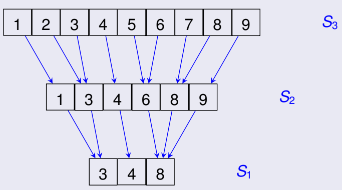
Fractional cascading

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 $S_1, \dots, S_m$. 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

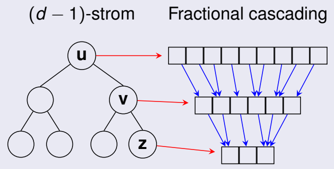
$\mathcal{O}(m + \log n)$

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.

Layered range trees

Using fractional cascading

- Every $(d-1)$-tree has assigned one fractional cascade instead of d-trees.
- Every element of the cascade has two pointers (for left and right children).



Complexity of Query

- Query in one $(d-1)$-tree takes $\mathcal{O}(\log n)$
- There are $\mathcal{O}\left(\log^{d-2} n\right)$ accessed $(d-1)$-trees in one Query
- Complexity of Query is $\mathcal{O}\left(k + \log^{d-1} n\right)$

Range trees: Further improvements

Using fractional cascading

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

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

Chazelle [2, 3]

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

Memory: $\mathcal{O}\left(n \left(\frac{\log n}{\log \log n}\right)^{d-1}\right)$

Chazelle, Guibas [4] pro $d \geq 3$

Query: $\mathcal{O}\left(k + \log^{d-2} n\right)$

Memory: $\mathcal{O}\left(n \log^{d} n\right)$

Outline

k-d trees

Description

- Points are stored in a binary tree
- In the root r, we store the median point m according to the first coordinate
- In the left subtree of r, we store all points having the first coordinate smaller than median
- In the right subtree of r, we store all points having the first coordinate larger than median
- Points in second level are split analogously by the second coordinate, etc.
- Points in i-th level are split are split by $i \mod d$ coordinate
- Height of the tree is $\log_2 n + \Theta(1)$
- Space complexity is $\mathcal{O}(n)$
- Complexity of operation Build is $\mathcal{O}(n \log n)$

k-d trees: Operation Query

Algorithm

```
Procedure Query (node v, range R)
    if v = NIL then
        return
    if v in R then
        Output: v
    if R is "left" from the point stored in v according to the levels' coordinate then
        Query (left child of v, R)
    else if R is "right" from the point stored in v according to the levels' coordinate then
        Query (right child of v, R)
    else
        Query (left child of v, R)
        Query (right child of v, R)
```


k-d trees: Complexity of operation Query

The worst-case example in $\mathbb{R}^2$

- Consider a set of points $S = \{(x, y); x, y \in [m]\}$ where $n = m^2$
- Consider a range $\langle 1, 2; 1, 8\rangle \times \mathbb{R}$
- In every level spliting by y coordinate both subtrees of every must be explored
- There are $\frac{1}{2}\log_2 n + \Theta(1)$ levels separating by y coordinates
- Total number of visited leaves is $2^{\frac{1}{2}\log_2 n + \Theta(1)} = \Theta(\sqrt{n})$

The worst-case example in $\mathbb{R}^d$

- Consider a set of points $S = [m]^d$ where $n = m^d$
- Consider a range $\langle 1, 2; 1, 8\rangle \times \mathbb{R}^{d-1}$
- In every level not spliting by the first coordinate, both subtrees of every must be explored
- Both subtrees are explored in all nodes of $\frac{d-1}{d}\log_2 n + \Theta(1)$ levels
- The total number of visited leaves is $2^{\frac{d-1}{d}\log_2 n + \Theta(1)} = \Theta(n^{1-\frac{1}{d}})$

Outline

[1] R Bayer and E McCreight.
Organization and maintenance of large ordered indexes.
Acta Informatica, 1:173–189, 1972.

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

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

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

[5] Michael L Fredman and Robert Endre Tarjan.
Fibonacci heaps and their uses in improved network optimization algorithms.
Journal of the ACM (JACM), 34(3):596–615, 1987.

[6] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran.
Cache-oblivious algorithms.
In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.

[7] Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts.

A new representation for linear lists.
In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60. ACM, 1977.

[8] Scott Huddleston and Kurt Mehlhorn.
A new data structure for representing sorted lists.
Acta informatica, 17(2):157–184, 1982.

[9] Donald E. Knuth.
Optimum binary search trees.
Acta informatica, 1(1):14–25, 1971.

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

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

[12] Anna Pagh, Rasmus Pagh, and Milan Ruzic.
Linear probing with constant independence.
In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318–327, 2007.

[13] Rasmus Pagh and Flemming Friche Rodler.

Cuckoo hashing.
Journal of Algorithms, 51(2):122–144, 2004.

[14] Mihai Pătraşcu and Mikkel Thorup.
On the k-independence required by linear probing and minwise independence.
In *International Colloquium on Automata, Languages, and Programming*, pages 715–726, 2010.

[15] Mihai Pătraşcu and Mikkel Thorup.
The power of simple tabulation hashing.
Journal of the ACM (JACM), 59(3):14, 2012.

[16] Jeanette P Schmidt and Alan Siegel.
The spatial complexity of oblivious k-probe hash functions.
SIAM Journal on Computing, 19(5):775–786, 1990.

[17] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan.
Chernoff-hoeffding bounds for applications with limited independence.
SIAM Journal on Discrete Mathematics, 8(2):223–250, 1995.

[18] Daniel D Sleator and Robert E Tarjan.
Amortized efficiency of list update and paging rules.
Communications of the ACM, 28(2):202–208, 1985.

[19] Daniel Dominic Sleator and Robert Endre Tarjan.
Self-adjusting binary search trees.

Journal of the ACM (JACM), 32(3):652–686, 1985.