Dictionary problem

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

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

Example of data structures
- Array
- Linked list
- Searching trees (e.g. AVL, red-black)
**Data Structures 1**

**Example: (2,4)-tree**

```
T
2 5 8
1 4 7 9
22 33 59
```

**Operation Find**

Search from the root using keys stored in internal nodes

**Time complexity**

Linear in height of the tree

---

**Algorithm (a,b)-tree: Insert**

1. Find the proper parent v of the inserted entity
2. Add a new leaf into v
3. while deg(v) > b do
   1. Find parent u of node v
   2. if v is the root then
      1. Create a new root with v as its only child
      else
         1. v ← parent of v
         2. Split node v into v and v'
   3. Create a new child v' of u immediately to the right of v
   4. Move the rightmost (b + 1)/2 children of v to v'
   5. v ← u

---

**Algorithm (a,b)-tree: Delete**

1. Find the leaf l containing the deleted key
2. v ← parent of l
3. Delete l
4. while deg(v) < a and v is not the root do
   1. u ← an adjacent sibling of v
   2. if deg(u) > a then
      1. Move the proper child from u to v
      else
         1. Move all children of u to v
         2. if v has no sibling then
            1. Remove the root (parent of v) and make v the new root
         else
            1. v ← parent of v

---

**Algorithm (a,b)-tree: Join**

**Description**

Union of two (a,b)-trees T₁ and T₂ assuming max key(T₁) < min key(T₂).

**Algorithm**

1. if height(T₁) ≥ height(T₂) then
   1. u ← last node of T₁ in height height(T₁) − height(T₂)
   2. v ← root of T₂
   else
      1. u ← last node of T₂ in height height(T₂) − height(T₁)
      2. v ← root of T₁
      3. Move all children of v to u
      4. if deg(u) > b then
         1. Recursively split u like in the operation insert

**Complexity**

Linear in the difference of heights of trees.

---

**Algorithm (a,b)-tree: Split**

**Description**

Given an (a,b)-tree T and a key x, split T to two (a,b)-trees T₁ and T₂ with keys smaller and greater than x, respectively.

**Algorithm (only for T₁)**

Input: (a,b)-tree T, key x
1. Q₀ ← an empty stack
2. t ← the root of T
3. while t is not a leaf do
   1. v ← child of t according to the key x
   2. Push all left brothers of v to Q₀
   3. t ← v
4. t₀ ← an empty (a,b)-tree
5. while Q₀ is non-empty do
   1. t₁ ← JOIN(POP(Q₀), T₀)

**Time complexity**

O(\log n) since complexity of JOIN is linear in the difference of heights of trees.
Description
Returns the $i$-th smallest key in the tree for given $i$.

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

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

Assumption
$b \geq 2a$

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

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

A-sort
Goal
Sort “almost sorted” list $x_1, x_2, \ldots, x_n$.

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

Idea: Insert $x_i = 16$ to this subtree
Insert $x_i = 16$ to this subtree

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

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

A-sort: Applications
- A-sort
- File systems e.g. Ext4, NTFS, HFS+, FAT
- Databases
Red-black tree: Definition

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

Example

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

Equivalence to (2,4)-tree
- Recolour the root to be black
- Combine every red node with its parent

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

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

Red-black tree: Insert

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

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

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

Red-black tree: Insert — uncle is red

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Worst-case</th>
<th>Amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incrementing a binary counter</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Insert into a dynamic array</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Multi-pop from a stack</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Methods**

- Aggregate analysis
- Accounting method
- Potential method

**Splay tree:** Splay a given node $x$

- Zig step: If the parent $p$ of $x$ is the root
  
  ![Zig step diagram]

- Zig-zig step: $x$ and $p$ are either both right children or are both left children
  
  ![Zig-zig step diagram]

- Zig-zag step: $x$ is a right child and $p$ is a left child or vice versa
  
  ![Zig-zag step diagram]

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

**Splay tree: Amortized time**

Lemma

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

**Notations**

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

**Splay tree: Zig step**

From the third point follows $\frac{s'(-p)}{s(p)} + \frac{s'(g)}{s(g)} \leq -2$, so we use the lemma to obtain

$$\log_2 s'(p) + \log_2 s'(g) \leq 2 \log s'(x) - 2.$$

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

**Splay tree: Zig-zag step**

Observe

- $r'(x) - r(g)$
- $r(x) < r(p)$
- $s'(p) + s'(g) \leq s'(x)$
- $r'(p) + r'(g) \leq 2r'(x) - 2$
- $\Delta \Phi = r'(g) - r(g) + r'(p) - r(p) + r'(x) - r(x) \leq 2(r'(x) - r(x)) - 2$
**Splay tree: Zig-zig step**

![Splay tree diagram](image)

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

**Splay tree: Insert**

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

**Amortized complexity**
- Find and splay: \( O(\log n) \)
- The potential \( \Phi \) is increased by at most \( r(x) + r(u) \leq 2 \log n \)

**Outline**
- (a,b) tree
- Red-black tree
- Splay tree
- Heaps
  - d-ary heap
  - Binomial heap
  - Lazy binomial heap
  - Fibonacci heap
  - Dijkstra's algorithm
- Cache-oblivious algorithms
- Hash tables
- Geometry
- Bibliography

**d-ary heap**

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

**Binary heap**

- Binary heap is a 2-ary heap

**Example of a binary heap**

![Binary heap example](image)

**Binary heap stored in a tree**

![Binary heap tree](image)

**Binary heap stored in an array**

![Binary heap array](image)
### d-ary heap: Height of the tree

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

### d-ary heap: Insert and decrease key

#### Insert: Algorithm

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

#### Decrease key (of a given node)

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

#### Complexity

$O(\log n)$

### d-ary heap: Building

#### Goal

Initialize a heap from a given array of elements

#### Algorithm

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

#### Correctness

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

### 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, ..., $k - 1$.
- 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

- Let $B_k$ be the binomial heap.
- $B_k$ contains $2k + 1$ trees, each of order $0, 1, ..., k - 1$.
- $B_k$ has $2k + 1$ leaves.
- $B_k$ has $2k - 1$ non-leaf nodes.

### Binomial tree: Example

#### Recursions for binomial heaps

- For $k = 1$, $B_1$ consists of $B_0$ and $B_1$.
- For $k = 2$, $B_2$ consists of $B_0$, $B_1$, and $B_2$.
- For $k = 3$, $B_3$ consists of $B_0$, $B_1$, $B_2$, and $B_3$.

### Binomial tree: Example

#### Recursions for binomial heaps

- For $k = 1$, $B_1$ consists of $B_0$ and $B_1$.
- For $k = 2$, $B_2$ consists of $B_0$, $B_1$, and $B_2$.
- For $k = 3$, $B_3$ consists of $B_0$, $B_1$, $B_2$, and $B_3$.

### Binomial tree: Example

#### Recursions for binomial heaps

- For $k = 1$, $B_1$ consists of $B_0$ and $B_1$.
- For $k = 2$, $B_2$ consists of $B_0$, $B_1$, and $B_2$.
- For $k = 3$, $B_3$ consists of $B_0$, $B_1$, $B_2$, and $B_3$.

### Binomial tree: Example

#### Recursions for binomial heaps

- For $k = 1$, $B_1$ consists of $B_0$ and $B_1$.
- For $k = 2$, $B_2$ consists of $B_0$, $B_1$, and $B_2$.
- For $k = 3$, $B_3$ consists of $B_0$, $B_1$, $B_2$, and $B_3$.

### Binomial tree: Example

#### Recursions for binomial heaps

- For $k = 1$, $B_1$ consists of $B_0$ and $B_1$.
- For $k = 2$, $B_2$ consists of $B_0$, $B_1$, and $B_2$.
- For $k = 3$, $B_3$ consists of $B_0$, $B_1$, $B_2$, and $B_3$. 
Binomial tree: Properties

Recursions for binomial heaps

Observations

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

Example of a set of binomial trees on 1010 nodes

Binomial heap: Representation

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

Binomial trees in a binomial heap

Binomial trees are stored in a linked list.

Remarks

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

Binomial heap: Operations Join and Insert

Join

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

Example

<table>
<thead>
<tr>
<th>Binomial tree</th>
<th>$B_0$</th>
<th>$B_1$</th>
<th>$B_2$</th>
<th>$B_3$</th>
<th>$B_4$</th>
<th>$B_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>First binomial heap</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Second binomial heap</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Joined binomial heap</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Complexity of operation Insert

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

Insert

Insert is implemented as join with a new tree of order zero.
- The worst-case complexity is $O(\log n)$.
- The amortized complexity is $O(1)$ — likewise increasing a binary counter.

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

<table>
<thead>
<tr>
<th>Binary number</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binomial heap contains:</td>
<td>$B_7$</td>
<td>$B_6$</td>
<td>$B_5$</td>
<td>$B_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example of a set of binomial trees on 1010$_2$ nodes

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

<table>
<thead>
<tr>
<th>Binary number</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binomial heap contains:</td>
<td>$B_7$</td>
<td>$B_6$</td>
<td>$B_5$</td>
<td>$B_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Binomial heap: Operations Decrease-key and Simple join

Decrease-key

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

Simple join

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

The following values need to be set:
- the child pointer in the node $u$,
- the parent and the sibling pointers in the node $v$ and
- the number of children in the node $u$.

Binomial heap: Operations Find-min and Delete-min

Find-min

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

Delete-min

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

Example
Lazy binomial heap

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

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

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

Lazy binomial heap: Reconstruction to the proper binomial heap

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

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

Fibonacci heap

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

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

Lazy binomial heap: Decrease-key

Example
```
1  1  4  3  6  7
A B A C D E F
```

Algorithm
```
Input: A node $u$ and new key $k$
1. Decrease key of the node $u$
2. If $u$ is a root or the parent of $u$ has key at most $k$ then
3. Return $u$
4. The minimal heap property is satisfied
5. $p$ ← the parent of $u$
6. Unmark the flag in $u$
7. Remove $u$ from its parent $p$ and append $u$ to the list of heaps
8. While $p$ is not a root and the flag in $p$ is set do
9. $u$ ← $p$
10. $p$ ← the parent of $u$
11. Unmark the flag in $u$
12. Remove $u$ from its parent $p$ and append $u$ to the list of heaps
13. If $p$ is not a root then
14. Set the flag in $p$
```

Unique 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
```
1. Initialize an array of pointers of size $\lceil \log_2 n \rceil$
2. For each heap $h$ in the lazy binomial heap do
3. $o$ ← order of $h$
4. While array[$o$] is not NIL do
5. $h$ ← the join of $h$ and array[$o$]
6. array[$o$] ← NIL
7. $o$ ← $o + 1$
8. Array[$o$] ← $h$
9. Create a binomial heap from the array
```
Data Structures

Fibonacci heap: Delete-min

Algorithm
Input: A node $u$ to be deleted
1. Delete the node $u$ and append its children to the list of trees
2. Reconstruction likewise in lazy binomial heap
3. Initialize an array of pointers of a sufficient size
4. for each tree $t$ in the Fibonacci heap do
5.   $c ←$ the number of children of the root of $t$
6.   while array[$c$] is not NIL do
7.     $t ←$ the join of $t$ and array[$c$]
8.     $c ← c + 1$
9.     $array[c] ← t$
10. Create a Fibonacci heap from the array

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 $s_0 ≥ s_1 + s_2 + s_3 + s_4 + \cdots + s_k + s_{k+1} + s_k + 1$.

Example

Fibonacci heap: Complexity

Worst-case complexity
- Operation Insert: $O(1)$
- Operation Decrease-key: $O(\log n)$
- Operation Delete-min: $O(n)$

Amortized complexity: Potential
$\Phi = \frac{3}{2} + 2m$ where $l$ is the number of trees and $m$ is the number of marked nodes

Amortized complexity: Insert
- cost: $O(1)$
- $\Delta\Phi = -1$
- Amortized complexity: $O(1)$

Amortized complexity: Delete-min
- cost: $O(\log n)$
- $\Delta\Phi ≤ O(\log n)$
- Amortized complexity: $O(\log n)$

Remaining parts
- Cost: $O(\log n)$
- $\Delta\Phi = 0$
- Amortized complexity: $O(\log n)$

Total amortized complexity
$O(\log n)$

Appendix

Fibonacci heap: Fibonacci numbers

Definition
- $F_0 = 0$
- $F_1 = 1$
- $F_k = F_{k-1} + F_{k-2}$ for $k ≥ 2$

Properties
- $\sum_{k=1}^{n} F_k = F_{n+2} - 1$
- $F_k = \omega_2^{\lfloor \log_5 F_k \rfloor}$
- $F_k ≥ \left(\frac{\sqrt{5} - 1}{2}\right)^{k+2}$

Proof
A straightforward application of the mathematical induction.

Fibonacci heap: Amortized complexity of Decrease-key

Single iteration of the while-loop (unmark and cut)
- Cost: $O(1)$
- $\Delta\Phi = -1 - 2 = -1$
- Amortized complexity: Zero

Remaining parts
- Cost: $O(1)$
- $\Delta\Phi ≤ 1$
- Amortized complexity: $O(1)$

Total amortized complexity
$O(1)$
Data Structures 1 82

Complexity table

<table>
<thead>
<tr>
<th></th>
<th>Binary</th>
<th>Binomial</th>
<th>Lazy binomial</th>
<th>Fibonacci</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>worst</td>
<td>worst</td>
<td>worst</td>
<td>worst</td>
</tr>
<tr>
<td></td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
<tr>
<td>Decrease-key</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
<tr>
<td>Decrease-min</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
<td>log n</td>
</tr>
</tbody>
</table>

Bibliography

Red-black tree
Heaps
Splay tree
(a,b)-tree
k-ary
log
Binomial
Binary
log
m
1
k
log
log
m
1
k
log
log
log
Fibonacci
1
m
log
n
Binomial
log
1
log
log
1

Memory models: A trivial program

For simplicity, consider only two types of memory called a disk and a cache.

Memory is split into pages of size B.

The size of the cache is M, so it can store \( P = \frac{M}{B} \) pages.

CPU can access data only in cache.

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

For simplicity, the size of one element is unitary.

External memory model

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

Cache-oblivious model

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

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

Cache is assumed to be fully associative.

Heap: Overview

Techniques for memory hierarchy

Example of sizes and speeds of different types of memory

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Speeds</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache</td>
<td>32 KB</td>
<td>267 GB/s</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 KB</td>
<td>96 GB/s</td>
</tr>
<tr>
<td>L3 cache</td>
<td>8 MB</td>
<td>62 GB/s</td>
</tr>
<tr>
<td>RAM</td>
<td>2 GB</td>
<td>23 GB/s</td>
</tr>
<tr>
<td>HHD 1</td>
<td>112 GB</td>
<td>56 MB/s</td>
</tr>
<tr>
<td>HHD 2</td>
<td>2 TB</td>
<td>14 MB/s</td>
</tr>
<tr>
<td>Internet</td>
<td>∞</td>
<td>10 MB/s</td>
</tr>
</tbody>
</table>

A trivial program

Memory models

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b) tree</td>
<td></td>
</tr>
<tr>
<td>Red-black tree</td>
<td></td>
</tr>
<tr>
<td>Splay tree</td>
<td></td>
</tr>
<tr>
<td>Heaps</td>
<td></td>
</tr>
<tr>
<td>Cache-oblivous algorithms</td>
<td></td>
</tr>
<tr>
<td>Hash tables</td>
<td></td>
</tr>
<tr>
<td>Geometry</td>
<td></td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
</tr>
</tbody>
</table>

Heaps: Dijkstra’s algorithm

Problem

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

Algorithm

1. Create an empty priority queue \( Q \) for vertices of \( G \)
2. for \( v \in V \) do
   1. \( \text{distance}(v) = 0 \) if \( v = s \) else \( \infty \)
   2. Insert \( v \) with the key \( \text{distance}(v) \) into \( Q \)
3. while \( Q \) is non-empty do
   1. Extract the vertex \( u \) with the smallest key (distance) from \( Q \)
   2. for \( v \leftarrow \text{neighbour of } u \) do
      1. if \( \text{distance}(v) > \text{distance}(u) + \omega(u, v) \) then
         1. \( \text{distance}(v) \leftarrow \text{distance}(u) + \omega(u, v) \)
         2. Decrease the key of \( v \) in \( Q \)

Heaps: Dijkstra’s algorithm

Number of operations

Dijkstra’s algorithm may call

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

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

Complexity table

| Array | Linear | Binomial | Lazy binomial | Fibonacci | k-ary |
|-------|--------|----------|---------------|-----------|
| Insert| Array  | Linear   | Binomial      | Lazy binomial | Fibonacci | k-ary |
|       | Array  | Linear   | Binomial      | Lazy binomial | Fibonacci | k-ary |
|       | Array  | Linear   | Binomial      | Lazy binomial | Fibonacci | k-ary |
| Delete-min | Array  | Linear   | Binomial      | Lazy binomial | Fibonacci | k-ary |
| Decrease-key | Array  | Linear   | Binomial      | Lazy binomial | Fibonacci | k-ary |
| Dijkstra’s | Array  | Linear   | Binomial      | Lazy binomial | Fibonacci | k-ary |

Linear-time complexity

- When \( m = n \) using an array.
- When \( m = \Omega(n^2) \) using a (a,b)-heap.
- When \( m = \Omega(\log n) \) using a Fibonacci heap.

Outline

1. (a,b)-tree
2. Red-black tree
3. Splay tree
4. Heaps
5. Cache-oblivious algorithms
6. Hash tables
7. Geometry
8. Bibliography

Memory models: A trivial program

For simplicity, consider only two types of memory called a disk and a cache.

Memory is split into pages of size \( B \).

The size of the cache is \( M \), so it can store \( P = \frac{M}{B} \) pages.

CPU can access data only in cache.

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

For simplicity, the size of one element is unitary.

External memory model

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

Cache-oblivious model

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

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

Cache is assumed to be fully associative.
Also called a block or a line.

For simplicity, we consider only loading pages from disk to cache, which is also called page faults.

Therefore, $B$ and $M$ are the maximal number of elements in a page and cache, respectively.

Half cache is for two input arrays and the other half is for the merged array.

Merging all blocks in level $i$ 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.

Funnelsort requires $O(\frac{n}{2}\log_2 n)$ page transfers.

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

One page stores $B$ nodes, so the one page stores a tree of height $\log_2(B) + O(1)$.

More precisely: $\Theta(\max\{1, \log n - \log B\})$.

---

### Cache-oblivious analysis: Scanning

**Scanning**

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

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

**Array reversal**

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

---

### Cache-oblivious analysis: Mergesort

**Case $n \leq M/2$**

Whole array fits into cache, so $2n/B + O(1)$ page are transferred. ☺

**Schema**

<table>
<thead>
<tr>
<th>Size of a block</th>
<th>Height of the recursion tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n/2$</td>
<td>$\log_2(n/2)$</td>
</tr>
<tr>
<td>$n/4$</td>
<td>$\log_2(n/4)$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\log_2 n$</td>
</tr>
<tr>
<td>$z$</td>
<td>$\log_2 z$</td>
</tr>
</tbody>
</table>

**Case $n > M/2$**

- Let $z$ be the maximal block in the recursion that can be sorted in cache.
- Observe: $\frac{n}{2} \leq z < 2z$.
- Merging one level requires $2^z + 2^z + O(1) = O(2^z)$ page transfers. ☺
- Hence, the number of page transfers is $O(2^z + 1) = O(2^z \log_2 n)$. ☺

---

### Cache-oblivious analysis: Binary heap and search

**Binary heap: A walk from the root to a leaf**

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

**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 transferred.

---

### Cache-oblivious analysis: Cache-aware search

**Search in a balanced binary search tree**

- Height of a tree is $\Theta(\log n)$, so $\Theta(\log n)$ pages are transferred. ☺

**Cache-aware algorithm**

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

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

- Choose $a$ and $b$ so that the size of one node of an $(a,b)$-tree is exactly $B$.
- Height of the $(a,b)$-tree is at most $\log_2 n + O(1)$.
- Search from the root to a leaf requires only $\Theta(\log_2 n)$ page transfers. ☺
- Replace every node of the $(a,b)$-tree by a binary subtree stored in one memory page. ☺
- A search in this binary tree requires also $\Theta(\log_2 n)$ page transfers. ☺
- However, we would prefer to be independent on $B$. ☺
When nodes are allocated independently, nodes on a path from the root to a leaf can be stored in different pages.

The height will be between \( \log_2 n \) and \( 1 + \log_2 n \) and these bounds would be equal to \( \Theta(\log_2 n) \).

Assuming whole subtree also fits into a single memory page.

This is also the best possible (the proof requires Information theory).

---

### 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**

```
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
```

- What is the number of subtrees?
- What is the number of nodes in each subtree?
- Is there a simple formula to determine indices of the parent and children for a given index of an element?
- Find algorithm which returns indices of the parent and children for a given index of an element.

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

```
Function Init(A, n, root_parent) ⬤
L ← empty
if n == 1 then
    A[0].parent ← root_parent
    A[0].children[0], A[0].children[1] ← NULL
else
    k ← min\(\lfloor \log_2 n \rfloor \) such that \( 2^k > n \) ⬤
    s ← \( 2^{k - 1} \) ⬤
    P ← Init(A, s, root_parent) ⬤
    C ← A + s ⬤
    i ← 0 ⬤
    while C < A + n do
        L.append(Init(C, min\(\lfloor (A + s + C) / 2 \rfloor \), P + s)) ⬤
        P[i/2].children[mod 2] ← C
        C ← C + s ⬤
        i ← i + 1
    return L
```

**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**

```
for i ← 0 to k do
    for j ← 1 to k do
        Swap(A[i], A[j])
```

**Assumptions**
For simplicity, we assume that \( B < k \) and \( P < k \).

**The number of page transfers by the simple algorithm**
- **Optimal page replacement**: \( \Omega((k - P)^2) \)
- **LRU or FIFO**: \( \Omega(k^2) \)
One page stores at most one row of the matrix and cache cannot store all elements of one column at once.

Cache-oblivious analysis: Matrix transposition: Simple approach

Representation of a matrix $5 \times 5$ in memory and an example of memory pages

<table>
<thead>
<tr>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
</tbody>
</table>

Optimal page replacement

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

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 transferred.

Cache-oblivious analysis: Matrix transposition: Optimal approach

Idea

Recursively split the matrix into sub-matrices:

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

Number of page transfers

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

Cache-oblivious analysis: Comparison of LRU and OPT strategies

**Theorem (Sleator, Tarjan, 1985)**

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

Then,

$$F_{lru} \leq \frac{n_{lru}}{n_{lru} + n_{opt} + 1} F_{opt} + n_{opt}.$$  

**Corollary**

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

The asymptotic number of page faults for some algorithms

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

- Scanning: $O(n/B)$
- Mergesort: $O(\log_2 n)$
- Funnelsort: $O(\log \log n)$
- The van Emde Boas layout: $O(\log \log n)$

Cache-oblivious analysis: Other algorithms and data structures

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

Outline

- (a,b) tree
- Red-black tree
- Splay tree
- Heaps
- Cache-oblivious algorithms
- Hash tables
  - Separate chaining
  - Linear probing
  - Cuckoo hashing
  - Hash functions
- Geometry
- Bibliography
Hash tables

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

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

Birthday paradox
When $n$ balls are (uniformly and independently) thrown into $m \geq n$ bins, the probability that every bin has at most one ball is
$$\prod_{i=1}^{m} \left(1 - \frac{1}{m}\right) \approx e^{-\frac{n^2}{2m}}.$$  

Hash tables: Separate chaining

Description
- Bucket $i$ stores all elements $x \in S$ with $h(x) = i$ using some data structure, e.g.
  - a linked list
  - a dynamic array
  - a self-balancing tree

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

Hash tables: Separate chaining: Analysis

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

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

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

Hash tables: Separate chaining: Example

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

<table>
<thead>
<tr>
<th>$x$</th>
<th>$h(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

- 37% buckets are empty for $\alpha = 1$
- Successful search: The total number of comparison to find all elements in the table is computed by summing over all buckets the number of comparisons to find all elements in a bucket, that is $\sum_{i=0}^{m-1} A_i$. Hence, the expected number of comparisons is $\frac{\sum_{i=0}^{m-1} A_i^2}{\sum_{i=0}^{m-1} A_i} = \frac{n}{\alpha} = \frac{m}{\alpha}$. We apply Chernoff bound on variables $\ell_i$ to obtain $P[A_i > c] < e^{-c \ln(1 + c)/\ln(1 + \frac{m}{n})}$.

Let $\epsilon > 0$ and $c = (1 + \epsilon) \frac{\ln(1 + \frac{m}{n})}{\ln(1 + \frac{m}{n})}$. We have to estimate $P[\max A_i > c]$. Observe that $P[\max A_i > c] < \sum_i P[A_i > c] = mP[A_i > c]$. We apply Chernoff bound on variables $\ell_i$ to obtain $P[A_i > c] < e^{-c \ln(1 + c)/\ln(1 + \frac{m}{n})}$.

Cheat Bound
Suppose $X_1, \ldots, X_n$ are independent random variables taking values in $(0, 1)$. Let $X$ denote their sum and let $\mu = E[X]$ denote the sum’s expected value. Then for any $c > 1$ holds $P[X > \mu c] < e^{-\frac{c-1}{2}}$. Indeed, both $\frac{1}{\ln(1 + c)}$ and $\frac{\ln(1 + c)}{\ln(1 + \frac{m}{n})}$ converge to zero, so for sufficiently large $n$ the power of $n$ is negative. Hence, $P[\max A_i > (1 + \epsilon) \frac{\ln(1 + \frac{m}{n})}{\ln(1 + \frac{m}{n})}] < 1 - \frac{1}{n^2}.$
Hash tables: Separate chaining: Example

- If log \( A_i \) - k elements fall into the bucket \( j \) and others \( n-k \) elements falls into other buckets. Therefore,

\[
P[A_i = k] = \left( \frac{n}{m} \right) \left( 1 - \frac{1}{m} \right)^{n-k} \sim \frac{n^k}{k!} \left( 1 - \frac{a}{n} \right)^n \to e^{-a}
\]

- Let the indicator variable \( X \) be 1 if \( H_i \) is a bucket for \( i \in S \). The number of elements in \( T \) is

\[ X = \sum X_i \] with \( E[X] = E[\sum H_i] - [\sum E[A_i]] = \alpha \cdot d \cdot \log n. \]

Using Chernoff bound we obtain

\[
P[X > e\mu] < \exp(-\alpha \cdot d \cdot (c-1 - \log e)) = n^{-\alpha \cdot d}
\]

for \( c = e \).

- A sequence of \( \Theta(\log n) \) operations Find can be split into subsequences of log \( n \) length. Furthermore, we use a cache for log \( n \) elements to avoid repetitive searching of elements in the same bucket.

- If log \( n \) searched elements are chosen randomly, they belong to pair-wise different buckets with high probability (see the birthday paradox).

Hash tables: Separate chaining: Analysis

- The worst case search time for one element

\[
The expected time for operations Find in the worst case is \( O(\log \log n) \).
\]

- Goal

Amortized complexity of searching is \( O(1) \) with high probability.

- Probability of \( k \) comparisons for searching one element

\[
\lim_{n \to \infty} P[A_i = k] = \frac{1}{k!}
\]

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

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

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

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

Hash tables: Separate chaining: Multiple-choice hashing

- 2-choice hashing

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

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

The expected length of the longest chain is \( O(\log(\log n)) \).

- d-choice hashing

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

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

The expected length of the longest chain is \( \Theta(\log(\log n)) + O(1) \).

Hash tables: Linear probing

- Memory consumption for separate chaining

Separate chaining uses memory for \( n \) element and

- \( m + n \) pointers if buckets are implemented using linked list, or

- \( m \) pointers and \( m \) integers if buckets use dynamic arrays.

- Goal

Store elements directly in the table.

- Linear probing

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

assumed \( n \leq m \).

- Operation Find

Iterate until the given key or empty bucket is found.

- Operation Delete

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

- Let \( 0 < \alpha < 1 \) and \( 1 < c < \frac{1}{1-\alpha} \) and \( q = \left( \frac{1-\alpha}{1-\alpha} \right) \). Observe \( 0 < q < 1 \).

- First, we estimate the probability \( p_i \) that \( t \) elements of \( S \) are hashed into \( T \) for given subset of buckets \( T \) of size \( t \), that is \( p_i = P[H_i \cap T] = k \). In order to apply Chernoff bound, let \( X_i \) be the indicator variable that \( H_i \) in \( T \) for all \( i \in S \).

Then, \( \mu = t \alpha \). Hence, \( P[X > e\mu] < \exp(-\alpha \cdot d \cdot (c-1 - \log e)) = n^{-\alpha \cdot d} \).

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

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

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 \( h_1(x) + h_2(x) \mod m \) with minimal \( i \geq 0 \) where \( h_1 \) and \( h_2 \) are two hash functions.

- Brent’s variation for operation Insert

If the bucket

\[
b - h_1(y) \mod m \text{ is occupied by an element } y \text{ and}
\]

\[
b + h_2(y) \mod m \text{ is also occupied but}
\]

\[
c = b + h_2(y) \mod m \text{ is empty, then move element } y \text{ to } c \text{ and insert } x \text{ to } b. \text{ This reduces the average search time.}
Hash tables: Cuckoo hashing

Origin
Rasmus Pagh and Flemming Friche Rodler [6]

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

Insert: Example
- Successful insert of element \( x \) into \( h_1(x) \) after three reallocations.
- Impossible insert of element \( y \) into \( h_2(y) \).

\[ \begin{align*}
  \text{insert}(x) & \rightarrow h(x) \\
  h(x) & = \{a, c, e, f, h, l, k, m, h, b, r, s\} \\
  h(a) & \text{ or } h(b)
\end{align*} \]

Properties of the cuckoo graph
- Undirected cuckoo graph
  - 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 \( m \geq 2c \). For given positions \( i \) and \( j \), the probability that there exists a path from \( i \) to \( j \) and the shortest one has length \( k \) is at most \( \frac{1}{2c} \).

Complexity of operation Insert without rehashing
Let \( c > 1 \) and \( m \geq 2c \). The expected length of the path is \( O(1) \).

Number of rehashes
Let \( c > 2 \) and \( m \geq 2c \). The expected number of rehashes is \( O(1) \).

Proof of the lemma by induction on \( k \):
- Let \( k = 1 \). For one element, the probability that it forms an edge \( ij \) is \( \frac{1}{m} \). So, the probability that there is an edge \( ij \) at most \( \frac{1}{2c} \).
- Let \( k > 1 \). 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 \( ij \). For one position, \( u \), the \( i \)-u path exists with probability \( \frac{1}{m} \). The conditional probability that there exists the edge \( ij \) if there exists \( i \)-u path is at most \( \frac{1}{2c} \) 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 \( \frac{k}{2c} \).

Insert without rehashing:
- Using the previous lemma for all length \( k \) and all end vertices \( ij \), the expected length of the path during operation Insert is \( m \sum_{k=1}^{\infty} k \frac{1}{2c} = \frac{m}{2c(2c+1)} \).
- The probability that inserting \( k \) times is at most \( \frac{k}{2c(2c+1)} \).
- The expected number of rehashes is at most \( \sum_{k=1}^{\infty} \frac{k}{2c(2c+1)} = \frac{1}{2c} \).

Hash tables: Hash functions

Basic terms
- Universe \( U = \{0, 1, \ldots, u - 1\} \) of all elements
- Represent a subset \( S \subseteq U \) of size \( n \)
- Store \( S \) in an array of size \( m \) using a hash function \( h : U \rightarrow M \) where \( M = \{0, 1, \ldots, m - 1\} \).

Hashing random data
Every reasonable function \( f : U \rightarrow S \) is sufficient for hashing random data, e.g. \( f(x) = x \mod m \).

Random hash function
\( u \log m \) bits are necessary to represent a random hash function.

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

Hash tables: Universal hashing

Universal hashing
A set \( H \) of hash functions is universal if randomly chosen \( h \in H \) satisfies
\[
\forall x_1 \neq x_2 \in U \quad \forall i \quad P[h(x_i) = z_i] \leq \frac{1}{m}
\]

2-universal hashing
A set \( H \) of hash functions is 2-universal if randomly chosen \( h \in H \) satisfies
\[
P[h(x_i) = z_i \text{ and } h(x_j) = z_j] \leq \frac{1}{m^2}
\]

Relations
- If a function is \( k \)-universal, then it is also \( k - 1 \) universal.
- If a function is 2-universal, then it is also universal.
- 1-universal function may not be universal.
Consider $x_1, x_2, y \in [m]$ such that $y = \alpha x_1 + \beta x_2 \pmod{m}$. Then, $\beta$ divides $\alpha x_1 - x_2$. Since $\alpha$ and $\beta$ are relatively prime, $\beta$ divides $x_2 - x_1$ which implies $x_2 = x_1$.

**Hash tables: Universal hashing: Multiply-mod-prime**

**Definition**
- $p$ is a prime greater than $u$
- $h_u(x) = (ax + b \mod p) \mod m$
- $H = \{h_{ua}: a \in [1, \ldots, p-1], b \in [0, \ldots, p-1]\}$

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

\[
y_1 = ax_1 + b \mod p \\
y_2 = ax_2 + b \mod p
\]

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

**Univarsity**
The multiply-mod-prime set of functions $H$ is universal.

**Hash tables: Universal hashing: Multiply-mod-prime**

**Definition**
- $p$ is a prime greater than $u$
- $h_u(x) = (ax + b \mod p) \mod m$
- $H = \{h_{ua}: a \in [0, \ldots, p-1], b \in [0, \ldots, p-1]\}$

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

\[
y_1 = ax_1 + b \mod p \\
y_2 = ax_2 + b \mod p
\]

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

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

\[
P[h_u(x_1) = z_1 \land h_u(x_2) = z_2] \leq \left\lceil \frac{u}{p^2} \right\rceil
\]

So, the multiply-mod-prime set of functions $H$ is not 2-universal.

**Hash tables: Universal hashing: Multiply-shift**

**Bits selection**
For positive integers $a, b, x$, let $bit_{a,b}(x) = \left\lfloor \frac{x \mod 2^b}{2^a} \right\rfloor$.

**Multiply-shift**
- Assume $u = 2^a$ and $m = 2^t$.
- $h_u(x) = bit_{a+t-u}(ax)$
- $H = \{h_{ua}: a \text{ odd } w \text{-bit integer} \}$

**Example in C**

```c
uint64_t hash(uint64_t x, uint64_t a)
{
    return (a*x) >> (64-1);
}
```

**Univarsity (without a proof)**
For every $x_1, x_2 \in [2^w]$, $x_1 \neq x_2$ it holds $P[h_u(x_1) = h_u(x_2)] \leq \frac{2}{2^w}$

Consider $x_1, x_2, y \in [2^w]$ such that $y = \alpha x_1 + \beta x_2 \pmod{2^w}$. Then, $\beta$ divides $\alpha x_1 - x_2$. Since $\alpha$ and $\beta$ are relatively prime, $\beta$ divides $x_2 - x_1$ which implies $x_2 = x_1$.

**Hash tables: Universal hashing: Multiply-shift**

**Bits selection**
For positive integers $a, b, x$, let $bit_{a,b}(x) = \left\lfloor \frac{x \mod 2^b}{2^a} \right\rfloor$.

**Multiply-shift**
- Assume $u = 2^a$ and $m = 2^t$ and $v = w + l - 1$.
- $h_u(x) = bit_{a+t-u}(ax + b)$
- $H = \{h_{ua}: a, b \in [2^v]\}$

**Lemma**
If $\alpha$ and $\beta$ are relatively prime, then $x \rightarrow \alpha x \pmod{\beta}$ is a bijection on $[\beta]$.

**2-universality**
$H$ is 2-universal, that is for every $x_1, x_2 \in [2^w]$, $x_1 \neq x_2$ and $z_1, z_2 \in M$ it holds

\[
P[h_u(x_1) = z_1 \land h_u(x_2) = z_2] \leq \frac{2}{2^w}
\]

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

$H = \{x \rightarrow bit_{a+l-u}(ax + b); a, b \in [2^v]\}$ is 2-universal where $\nu = w + l - 1$

Let $s$ be the index of the least significant 1-bit in $(x_1 - x_2)$
Let $\nu$ be the odd number such that $x_2 - x_1 = 2^{\nu}$
- $a \rightarrow a \cdot \nu \mod 2^s - \text{bit}_{2^{\nu-u}}(a \cdot \nu)$ is a bijection on $[2^s]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot 2^s) - \text{bit}_{2^{\nu-u}}(a \cdot 2^{\nu-s})$ is a bijection on $[2^s]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot 2^s) - \text{bit}_{2^{\nu-u}}(a \cdot (x_1 - x_2))$ is a 2s-to-1 mapping $[2^s] \rightarrow [2^{\nu-s}]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot 2^s) - \text{bit}_{2^{\nu-u}}(a \cdot (x_1 - x_2))$ is a 2s-to-1 mapping $[2^s] \rightarrow [2^{\nu-s}]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot (x_1 - x_2))$ is a 2s-to-1 mapping $[2^s] \rightarrow [2^{\nu-s}]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot (x_1 - x_2))$ is a 2s-to-1 mapping $[2^s] \rightarrow [2^{\nu-s}]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot (x_1 - x_2))$ is a 2s-to-1 mapping $[2^s] \rightarrow [2^{\nu-s}]$.
- $a \rightarrow \text{bit}_{2^{\nu-u}}(a \cdot (x_1 - x_2))$ is a 2s-to-1 mapping $[2^s] \rightarrow [2^{\nu-s}]$.

If $a$ and $b$ are independently uniformly distributed on $[2^v]$, then

\[
P(h_u(x_1) = z_1 \land h_u(x_2) = z_2) = \frac{1}{2^{2v-1}}.
\]
1. Follows from lemma for \( \alpha - \beta \) and \( \beta = 2^k \).
2. The second equality uses \( \binom{a}{x} = 0 \).
3. Since \( (\alpha, \beta) \Rightarrow (\alpha, \alpha + \beta) \) is a bijection.

\[ x_0, \ldots, x_d \text{ are coefficients of a polynomial of degree } d. \]

\[ \text{Two different polynomials of degree at most } d \text{ have at most } d+1 \text{ common points, so there are at most } d+1 \text{ colliding values } x_i. \]

### Geometry: Range query in \( \mathbb{R}^1 \)

#### Example of 1D range tree

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

![Example of 1D range tree](image)

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

### Hash tables: Universal hashing: Multiplicative-shift for vectors

**Multiplicative-shift for fix-length vectors**

- Hash a vector \( x_0, \ldots, x_d \in U = [2^w] \) into \( S = [2^v] \) and let \( v \geq w + 1 \)
  - \( h_{\alpha}(x_0, \ldots, x_d) = \left( b + \sum_{i=1}^{d} a_i x_i \right) \mod p \)
  - \( H = \{ h_{\alpha} \mid \alpha \in [p] \} \)
  - \( P(h_{\alpha}(x_0, \ldots, x_d) = h_{\alpha}(y_0, \ldots, y_d)) \leq \frac{1}{p} \) for two different strings with \( d' < d \).

**Multiplicative-shift for variable-length string**

- Hash a string \( x_0, \ldots, x_d \in U \) into \([p] \) where \( p \geq u \) is a prime.
  - \( h_{\alpha}(x_0, \ldots, x_d) = \sum_{i=0}^{d} x_i a_i \mod p \)
  - \( H = \{ h_{\alpha} \mid \alpha \in [p] \} \)
  - \( P(h_{\alpha}(x_0, \ldots, x_d) = h_{\alpha}(y_0, \ldots, y_d)) \leq \frac{1}{p} \) for two different strings with \( d' < d \).

### Geometry

**Types of problems**

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

### Range query in \( \mathbb{R}^1 \)

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

- **Static**: Array; query in \( O(\log n) \)
- **Dynamic**: Balanced search tree; query and update in \( O(k + \log n) \)

### Nodes and leaves in every subtree.

- Nodes \( a \) and \( b \) are actually the successor of \( a \) and the predecessor of \( b \), respectively.
- Remember the number of leaves in every subtree.
Geometry: 2D range trees

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

Example

```
x-tree
  /
 /    /
V  y-trees
  /
W  /
  /
X
```

Contains the same points as the subtree of u.
Contains the same points as the subtree of w.
Contains the same points as the subtree of v.
Contains the same points as the subtree of z.

Et cetera.

Geometry: 2D range trees: Range query

Range query
- Search for keys $a$, and $b_i$ in the x-tree.
- Identify all inner nodes in the x-tree which store points with x-coordinate in the interval $(a_i, b_i)$.
- Run $(a_i, b_i)$-query in all corresponding y-trees.

Example

```

```

Complexity
$O(k \log^d n)$, since $(a_i, b_i)$-query is run in $O(\log n)$ y-trees.

Geometry: 2D range trees: Build

Straightforward approach
Create x-tree and then all y-trees using operation insert. Complexity is $O(n \log^d n)$.

Faster approach
First, create two arrays of points sorted by x and y coordinates. Then, recursively . . .
- Let x-root be the medium of all points by x-coordinate.
- Create y-tree for the x-root.
- Split both sorted arrays by x-root.
- Recursively create both children of x-root.

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

Geometry: 2D range trees: Space complexity

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

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

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

Geometry: 2D range trees: 3D range trees

3D range trees
- Create 2D range tree for x and y coordinates.
- For every node $u$ in every y-tree, create a search tree ordered z-coordinate containing all points of the subtree of $u$.

d-dimensional range trees
Add dimensions one by one likewise in 3D range tree.

```

```

Complexity
- Space: $O(n \log^{d-1} n)$ since every point is stored in $O(\log^d n)$ z-trees, etc.
- Query: $O(k \log^d n)$ since $(a_i, b_i)$-query is run in $O(\log^d n)$ z-trees, etc.
- Build: $O(n \log^d n)$ if dimension-trees are created one-by-one by insertion.
- $O(n \log^d n)$ if we use the faster approach likewise in 2D.

Geometry: Layered range trees

2D case
Replace y-trees by sorted arrays.

Example

```
x-tree
  /
 /    /
V  y-arrays
  /
W  /
  /
X
```

Et cetera.

Higher dimension
Replace trees of the last dimension by sorted arrays.
Motivational problem

Given sets $S_1, \ldots, S_n$, where $|S_i| = n$, create a data structure for fast searching elements $x \in S_i$ in all sets $S_1, \ldots, S_n$. 

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 $S_{i-1}$.

Complexity of a search in $m$ sets

$O(m \log n)$

Geometry: Fractional cascading

Using fractional cascading

For the last dimension arrays, e.g. $d = 2$:

- Search in the $x$-tree takes $O(\log n)$.
- Binary search for $a_i$ and $b_i$ in y-arrays takes $O(\log n)$.

Complexity of one range query in 2D

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


Description (Jiří Nierhoff, Edward M. Reingold [5])

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

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

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

Height

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

Amortized cost

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

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

Geometry: Range trees using $BB[i]$-trees

Dynamic range trees

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

2D case

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

3D case

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

$d$-dimensional range trees using $BB[i]$-trees

- Range query in $O\left(k + \log^d n\right)$ worst case.
- Insert and Delete in $O\left(\log^d n\right)$ amortized cost.
When we apply fractional cascading on leaves of a tree instead of arrays, we obtain query in $O\left(k + \log n \right)$ without changing the complexity for updates.

The actual time for $m$ updates is $O\left(n \log^{m-1} n + m \log^m n \right)$.

Geometry: Interval trees

Input
Set of intervals $S = \{(a_1, b_1), \ldots, a_k, b_k\}$ where $I_i = (a_i, b_i)$.

Recursive construction of interval trees
Interval tree is a binary tree. Let
1. $m$ be the medium of 2n endpoints $a_1, b_1, \ldots, a_n, b_n$.
2. $S_l = \{I_i | a_i \leq m \leq b_i\}$ contains intervals containing $m$.
3. $S_r = \{I_i | m < a_i \}$ be intervals smaller than $m$ and
4. $S_r = \{I_i | m < b_i \}$ be intervals greater than $m$.

The root of the tree contains
- two arrays of intervals $S$ sorted by left and right end-points,
- interval trees for intervals $S_l$ as the left child and
- interval trees for intervals $S_r$ as the right child.

Complexity
- Time complexity for construction is $O(n \log n)$.
- Space complexity is $O(n)$.

Geometry: Interval trees: Intersection interval query

Problem description
Given query interval $Q = (a, b)$, find all intervals intersecting with $Q$.

Recursive algorithm
1. If $a \leq m \leq b$ then
   1. Write all intervals $S_n$.
   2. Recursively process both children.
2. Else if $a < m$ then
   1. Use the array of intervals $S_a$ sorted by left end-points to find all intervals of $S_a$ intersecting with $Q$.
   2. Recursively process the left child.
3. Else
   1. Use the array of intervals $S_b$ sorted by right end-points to find all intervals of $S_b$ intersecting with $Q$.
   2. Recursively process the right child.

Complexity
$O(k \log n)$

Geometry: Segment trees

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

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

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

Time complexity of a query
$O(k \log n)$.

Geometry: Range trees: Further improvements

Bernard Chazelle [1, 2]
$d$-dimensional range query in $O\left(k + \log^d n \right)$ time and $O\left(\binom{d-1}{n} \right)$ space.

Bernard Chazelle, Leonidas J. Guibas [3]
$d$-dimensional range query in $O\left(k + \log^{d-1} n \right)$ time and $O\left(n \log^d n \right)$ space.

If $S_l$ or $S_r$ is empty, then there is no left or right child, respectively.

There are at most $n$ end-points smaller than $m$, so $S_l$ contains at most $\frac{n}{2}$ intervals. Therefore, the complexity satisfies the recurrence formula $T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$.

Every interval is stored in exactly one node. If $S_n$ is empty, then $n$ is even and both $S_l$ and $S_r$ contains $\frac{n}{2}$ intervals. There are at most $n - 1$ such nodes. Therefore, the tree has at most $2n - 1$ nodes.

Geometry: Segment trees

Input
Set of intervals $S = \{l_1, \ldots, l_k\}$ where $l_i = (a_i, b_i)$.

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

Trivial approach
1. Let $x_1, \ldots, x_n$ be sorted end-points $(a_1, b_1, \ldots, a_n, b_n)$ without duplicities.
2. Split $S$ into blocks $(-\infty, x_1), [x_1], [x_1, x_2), [x_2, \ldots, [x_n], (x_n, \infty)$.
3. For each block, store all intervals of $S$ containing the block.

Complexity
- Time for query: $O(k \log n)$
- Time for construction: $O(n^2)$
- Space: $O(n^2)$

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

Geometry: Priority search tree

Heap and search tree in one binary tree
If every element e has a key $k$ and a priority $p$ priority, it is possible to store a set of elements in a binary tree so that
- the min-heap property is satisfied for priorities and
- the search-tree property is satisfied for keys.

Relax the search tree property
Priority search tree is a binary tree having one element in every node so that
- the min-heap property is satisfied for priorities and
- elements can be found by their keys in $O(\log n)$ time.

Top-down recursive construction of a priority search tree
The root of the priority search tree storing a set of elements $S$ contains
- the element $e$ of $S$ with the smallest priority,
- the median key $m$ of all elements of $S$,
- the left subtree stores all elements with keys smaller than $m$ (except e) and
- the right subtree stores all elements with keys greater than $m$ (except e).
Observe that if all keys and all priorities are pair-wise different, then there exists a unique binary tree storing all elements.

Note that $m$ is not the key of the element $e$ (unless $e$ coincidentally has the median key).

Observe that this tree does not satisfy the search-tree condition in general.

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

1. **Geometry: Priority search tree**

2. **Complexity**
   - Space complexity is $O(n)$
   - Construction in $O(n \log n)$-time
   - Find the element with the smallest priority in $O(1)$-time
   - Find the element with a given key in $O(\log n)$-time
   - Delete the element with the smallest priority in $O(\log n)$-time

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

4. **Outline**
   - (a,b)-tree
   - Red-black tree
   - Splay tree
   - Heaps
   - Cache-oblivious algorithms
   - Hash tables
   - Geometry
   - Bibliography

---


