Self-Adjusting Binary Search Tree: the Splay Tree

Junhao Gan Tony Wirth

School of Computing and Information Systems
The University of Melbourne

March 28, 2022

Reading Materials

- The Seminal Paper by Daniel Dominic Sleator and Robert Endre Tarjan:
  https://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf
- Splay Tree by David Karger:
  http://courses.csail.mit.edu/6.854/17/Notes/n3-splay.html
- A Survey of Dynamic Optimality Conjecture of the Splay Tree by John Iacono:
  https://arxiv.org/pdf/1306.0207.pdf

The Splay Tree

About Binary Search Trees

In the previous lectures and also the exercises, we have introduced or discussed in detail on quite a number of binary search trees (BST's), such as:

- Red-Black Tree
- AVL Tree
- Weight-Balanced BST
- Randomized Treap

All of them aim to make the height of the tree as short as possible, equivalently, as "balanced" as possible. This is simply because, the cost of an operation depends on the depth of the target node.

About Binary Search Trees

To maintain the balance, all these trees have to store some **auxiliary information** in the nodes and maintain certain **balancing invariants**.

In this lecture, we introduce a surprisingly simple yet surprisingly powerful binary search tree, called the **Splay Tree**, where:

- no auxiliary information is needed to store in each node;
- no balancing invariant: the tree can be unbalanced;
- each of the operations: *insert*, *delete*, *search*, *split* and *join* can be performed in $O(\log n)$ amortized time.

In fact, the power of the Splay Tree is actually way beyond the $O(\log n)$ amortized bound.

Optimalities of the Splay Tree

Static Optimality

Consider a sufficiently long sequence, $\mathcal{L}$, of *successful search* operations *only* such that each element is accessed at least once.

The overall running time of a Splay Tree on $\mathcal{L}$ is as *good* (up to a constant factor) as the overall cost of *any* static tree on $\mathcal{L}$, where:

- a static tree is a tree that once it is constructed, its structure is fixed and cannot be changed.

Here, by *any* we mean, this also includes the optimal static tree designed with the search sequence $\mathcal{L}$ being given in advance.

Optimalities of the Splay Tree

Dynamic Optimality Conjecture

Even better, it is widely *conjectured* that the Splay Tree is actually dynamic optimal, that is, the overall cost of a Splay tree is at most a constant factor of the overall cost of the optimal dynamic tree for $\mathcal{L}$, where:

- an optimal dynamic tree for $\mathcal{L}$ is a tree:
  - its structure is allowed to be updated (e.g., by rotations) between the operations of $\mathcal{L}$, and
  - the overall cost (including the cost for updating the tree structure) is minimized.

Proving this conjecture is still a big open problem in the field of data structures, for more than 35 years.

Optimalities of the Splay Tree

More Optimalities

The Splay Tree is also optimal in terms of other metrics such as Static/Dynamic Finger Search Optimality and Cache Optimality. Please refer to the reading materials for more details.

It is widely believed that the Splay Tree indeed is the ultimate BST construction.

The Splay Tree

The Splay Tree is a binary search tree, where each element is stored in one and exactly one node.

Following the convention in this subject, we assume that the search keys of the elements are *distinct*.

There is a one-one-mapping between elements and nodes in a splay tree.

For simplicity, we use elements and nodes interchangeably, that is, we may use element x as the node in the tree containing x.

The Splay Tree

Essentially, the Splay Tree is a self-adjusting BST:

- a splay tree adjusts its tree structure automatically according to the operations met so far.

Basic Idea

If a node is accessed in the current operation, it is likely that it will be accessed again in the near future.

As such, it is sensible to adjust the tree such that the cost of the next access to this node can be reduced.

Recall that the cost for accessing a node x is $O(depth(x))$.

To reduce the next access cost of x, on accessing x, a splay tree **rotates** x **to be the root of the tree**.

The Splay Tree

Operations

- *splay*(x): rotate the node x to be the root of the tree;
- *insert*(x): insert x with the standard BST algorithm (insert x to a proper leaf node); *splay*(x);
- *delete*(x):
  - if x is the only node in the tree, delete x and done;
  - otherwise, swap x with a proper leaf node (according to the standard BST algorithm); let u be the parent of such leaf node; delete x and *splay*(u).
- *search*(x): search x with the standard BST algorithm; let u be the last node accessed in the search; *splay*(u).

The *split* and *join* operations are left as exercises.

The *splay*(x) operation is the only difference from a standard BST.

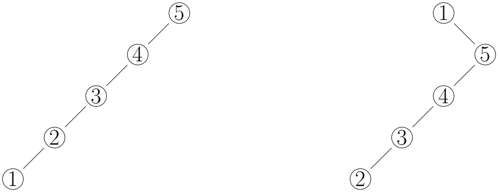
The *splay*(x) Operation

While there are multiple ways to rotate x to become the root of a tree, these have different effects.

The implementation of *splay*(x) is indeed very crucial.

Single Rotations

Rotate x with its parent until x becomes the root.



before and after splaying node 1 with single rotations

The height of the tree may not be reduced.

The $splay(x)$ Operation

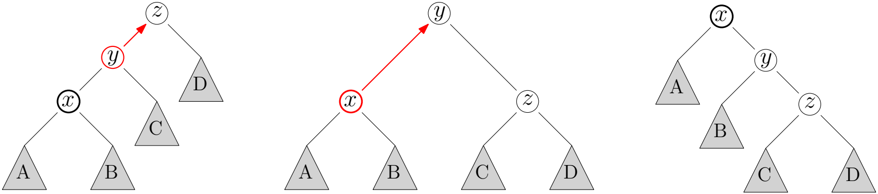
Instead of single rotations, the Splay Tree adopts double rotations.

Double Rotations

When performing double rotations, we consider the positions of both x and its parent y. There are six cases.

The Zig-Zig Case on x

y is the left (zig) child of z, and x is the left (zig) child of y.



first rotate y with z second rotate on x with y the outcome

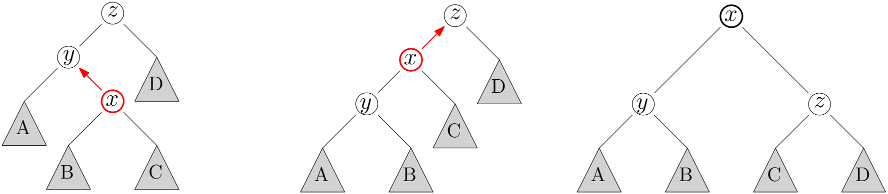
The **Zag-Zag** case on x is symmetric.

The *splay*(x) Operation

Double Rotations

The Zig-Zag Case on x

y is the left (zig) child of z, and x is the right (zag) child of y.



first rotate x with y — second rotate on x with z — the outcome

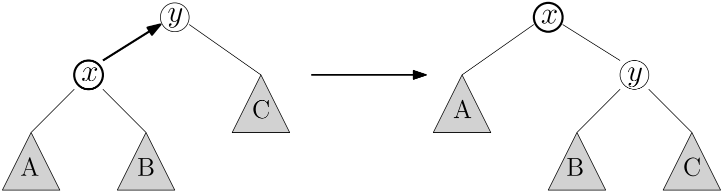
The **Zag-Zig** case on x is symmetric.

The *splay*(x) Operation

Double Rotations

The Zig Case on x

x is the left (zig) child of y, and y is the root of tree.



The **Zag** case on x is symmetric.

The $splay(x)$ Operation

The $splay(x)$ operation on x:

- repeatedly apply one of the above six cases to perform double rotations until x becomes the root of the tree.

Next, we show that the amortized cost of $splay(x)$ is $O(\log n)$.

We prove this bound with a potential function.

The Amortized Analysis

Let T be the splay tree, and $T(x)$ the sub-tree rooted at x.

For each node x in T, we define:

- a constant weight $w(x) > 0$;
- sub-tree weighted sum $s(x) = \sum_{u \in T(x)} w(u)$;
- rank $r(x) = \log_2 s(x)$.

We define the **potential function** $\Phi(S_i)$ as:

$$\Phi(S_i) = \sum_{x \in T} r(x).$$

At the current stage, we set $w(x) = 1$ for all $x \in T$. As a result,

- $s(x)$ is actually the sub-tree size rooted at x;
- $\Phi(S_0) = 0$ and $\Phi(S_i) \geq 0$ for all integer indexes $i \geq 1$.

The Amortized Analysis

We prove the following lemma:

The Access Lemma. The amortized cost of the *splay*(x) operation is at most

$$3 \cdot \left(r^{(1)}(x) - r^{(0)}(x)\right) + 1\,,$$

where $r^{(0)}(x)$ and $r^{(1)}(x)$ are the rank of x before and after the operation, respectively.

The Access Lemma holds for all constant $w(x) > 0$ for all $x \in T$.

The Amortized Analysis

A *splay* operation is essentially a sequence of double rotations.

To prove the lemma, it suffices to analyse the amortized cost of the double rotations in each of the six cases. More specifically, we shall show:

- for the **Zig** and **Zag** cases, the amortized cost is

$$\leq 3 \cdot (r_i^{(1)}(x) - r_i^{(0)}(x)) + 1;$$

- for all the other four cases, the amortized cost is

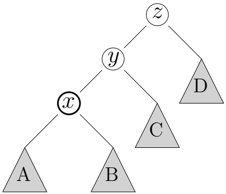
$$\leq 3 \cdot (r_i^{(1)}(x) - r_i^{(0)}(x));$$

where $r_i^{(0)}(x)$ and $r_i^{(1)}(x)$ are the ranks of x right before and right after the i-th double rotation in $splay(x)$.

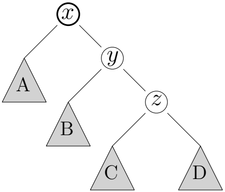
When the context is clear, we drop the index i from these notations.

We only show this for the **Zig-Zig** case (the harder case) here; and leave the other cases as exercises.

The Amortized Analysis



the Zig-Zig case



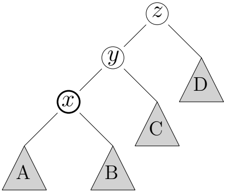
the outcome

Bounding $\Delta\Phi$

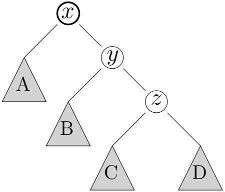
After the Zig-Zig double rotation, only the ranks of x, y and z would be changed, as their sub-tree sums have changed.

Thus, $\Delta\Phi = r^{(1)}(x) + r^{(1)}(y) + r^{(1)}(z) - r^{(0)}(x) - r^{(0)}(y) - r^{(0)}(z)$.

The Amortized Analysis



the Zig-Zig case



the outcome

Bounding $\Delta\Phi$

Observe that: (i) $r^{(1)}(x) = r^{(0)}(z)$; (ii) $r^{(0)}(y) \geq r^{(0)}(x)$; and (iii) $r^{(1)}(y) \leq r^{(1)}(x)$. We have:

$$\begin{aligned}\Delta\Phi &= r^{(1)}(x) + r^{(1)}(y) + r^{(1)}(z) - r^{(0)}(x) - r^{(0)}(y) - r^{(0)}(z) \\ &\leq r^{(1)}(x) + r^{(1)}(z) - r^{(0)}(x) - r^{(0)}(x) \\ &= r^{(1)}(x) + r^{(1)}(z) - 2 \cdot r^{(0)}(x) \qquad (1)\end{aligned}$$

The Amortized Analysis

Bounding $\Delta\Phi$

Since the log function is concave, $\frac{\log a + \log b}{2} \leq \log(\frac{a+b}{2})$. Let $s^{(0)}(x)$ and $s^{(1)}(x)$ be the sub-tree sums of a node x before and after the double rotation.

$$\begin{aligned}
\frac{r^{(0)}(x) + r^{(1)}(z)}{2} &= \frac{\log_2 s^{(0)}(x) + \log_2 s^{(1)}(z)}{2} \\
&\leq \log_2\left(\frac{s^{(0)}(x) + s^{(1)}(z)}{2}\right) \\
&\leq \log_2\left(\frac{s^{(1)}(x)}{2}\right) \qquad \left(\text{by } s^{(0)}(x) + s^{(1)}(z) \leq s^{(1)}(x)\right) \\
&= \log_2 s^{(1)}(x) - \log_2 2 \\
&= r^{(1)}(x) - 1
\end{aligned}$$

Therefore, we have:

$$r^{(1)}(z) \leq 2 \cdot r^{(1)}(x) - 2 - r^{(0)}(x) \tag{2}$$

The Amortized Analysis

Bounding $\Delta\Phi$

Substituting (2) to (1), we have:

$$\begin{aligned}\Delta\Phi &\le r^{(1)}(x) + r^{(1)}(z) - 2 \cdot r^{(0)}(x) \\ &\le r^{(1)}(x) + \left(2 \cdot r^{(1)}(x) - 2 - r^{(0)}(x)\right) - 2 \cdot r^{(0)}(x) \\ &= 3 \cdot \left(r^{(1)}(x) - r^{(0)}(x)\right) - 2\end{aligned}$$

Amortized Cost of Zig-Zig

The actual cost of the double rotation in the Zig-Zig case is 2. Thus,

$$\text{amortized cost} = 2 + \Delta\Phi \le 3 \cdot \left(r^{(1)}(x) - r^{(0)}(x)\right).$$

The Amortized Analysis

Amortized Cost of *splay*(x)

$$
\begin{aligned}
\text{amortized cost of } \mathit{splay}(x) &= \sum_i \text{amortized cost of the } i\text{-th double rotation} \\
&\le \sum_i 3 \cdot \left(r_i^{(1)}(x) - r_i^{(0)}(x) \right) + 1 \\
&\text{(the } +1 \text{ comes from the possible Zig or Zag case)} \\
&= 3 \cdot \left(r^{(1)}(x) - r^{(0)}(x) \right) + 1
\end{aligned}
$$

where $r^{(0)}(x)$ and $r^{(1)}(x)$ are the ranks of x before and after *splay*(x).

Therefore, the Access Lemma follows.

The Amortized Analysis

Amortized Cost of *splay*(x)

After *splay*(x), x will become the root of the tree.

With our setting of weights: $w(x) = 1$ for all $x \in T$, we have:

- $r^{(1)}(x) = \log_2 s^{(1)}(x) = \log_2 n$;
- $r^{(0)}(x) \geq \log_2 w(x) = 0$.

Substitute the above to the Access Lemma:

> **The Access Lemma.** The amortized cost of the *splay*(x) operation is at most $3 \cdot \left(r^{(1)}(x) - r^{(0)}(x)\right) + 1$.

The amortized cost of *splay*(x) is bounded by $3 \cdot (\log_2 n - 0) + 1 = O(\log n)$.

The Amortized Analysis

Remark

It can be verified that the above proof for the Access Lemma actually holds for any positive constant weight setting, i.e., for each x in T, $w(x)$ is a constant and $w(x) > 0$.

As we will see shortly, by choosing the weights $w(x)$ cleverly, we can derive different interesting optimality results.

The Amortized Analysis

In the following operation cost analysis, we set $w(x) = 1$ for all x in T.

Amortized Cost of *search*(x)

Observe that:

- the standard BST search algorithm does not change the potential;
- let u be the last node visited in the search; if x exists in T, then $u = x$.
- the actual cost of the standard BST search is the depth of u, i.e., the number of nodes along the path from *root* to u;
- *splay*(u) goes along the path from u up to *root*.

The actual searching cost can be charged to the actual cost of the *splay*(u) operation.

The Amortized Analysis

Amortized Cost of *search*(x)

Observe that after the charging, the actual cost of *splay*(u) effectively becomes twice of before.

By adjusting the constant factor in the potential function:

$$\Phi(S_i) \leftarrow 2 \cdot \Phi(S_i)\,,$$

and hence,

$$\Delta\Phi(S_i) \leftarrow 2 \cdot \Delta\Phi(S_i)\,.$$

Therefore, the amortized cost of *search*(x) is at most twice of the amortized cost of *splay*(u), and thus, still bounded by $O(\log n)$.

The Amortized Analysis

Amortized Cost of *insert*(x)

We show the amortized cost of *insert*(x) is bounded by $O(\log n)$.

Observe that the amortized cost of *insert*(x) can be calculated as the sum of the following four terms:

- the actual cost of inserting x with the standard BST algorithm;
- the change of potential before and after inserting x;
- the actual cost of *splay*(x);
- the change of potential before and after splaying x.

By an analogous argument in the analysis of *search*(x), we can charge the actual cost of *insert*(x) to the actual cost of the splay operation.

Thus, the sum of the three (expect the second) is bounded by $O(\log n)$.

It remains to show that the second term is also bounded by $O(\log n)$.

The Amortized Analysis

Amortized Cost of *insert*(x)

Let k be the depth of node x. Denote the nodes along the path from *root* to x by $y_k, y_{k-1}, \ldots, y_2, y_1$, respectively. In particular, $y_k = root$ and $y_1 = x$.

After inserting x, only the ranks of y_i (for $i = 1, \ldots, k$) have changed. The change of the potential before and after inserting x is:

$$\Delta\Phi = \sum_{i=1}^{k} \left(r^{(1)}(y_i) - r^{(0)}(y_i) \right) = \sum_{i=1}^{k} \left(\log s^{(1)}(y_i) - \log s^{(0)}(y_i) \right).$$

The Amortized Analysis

Amortized Cost of *insert*(x)

$$
\begin{aligned}
\Delta\Phi &= \sum_{i=1}^{k} \left(\log s^{(1)}(y_i) - \log s^{(0)}(y_i) \right) \\
&= \sum_{i=1}^{k} \left(\log(s^{(0)}(y_i) + 1) - \log s^{(0)}(y_i) \right) \\
&\leq \log(s^{(0)}(y_k) + 1) - \log s^{(0)}(y_k) + \sum_{i=1}^{k-1} \left(\log(s^{(0)}(y_{i+1})) - \log s^{(0)}(y_i) \right) \\
&= \log(s^{(0)}(y_k) + 1) - \log s^{(0)}(y_k) + \log s^{(0)}(y_k) - \log s^{(0)}(y_1) \\
&= \log(s^{(0)}(root) + 1) \qquad (\text{because } \log s^{(0)}(y_1) = \log s^{(0)}(x) = 0) \\
&= O(\log n)
\end{aligned}
$$

Therefore, from our earlier discussion, the amortized cost of *insert*(x) is bounded by $O(\log n)$.

The Amortized Analysis

Amortized Cost of *delete*(x)

As the procedure of *delete*(x) can be considered as a reverse of *insert*(x), the amortized analysis of *delete*(x) is analogous to that of *insert*(x).

We thus omit the analysis here and leave it as an exercise.

Static Optimality

Static Optimality

Successful-Search-Only Sequence

Consider a sufficiently long successful-search-only sequence $\mathcal{L}$ on a set P of elements, such that:

- for all *search*(x) operations, x is in the set P of the elements (i.e., the search is successful);
- each element in P is accessed (i.e., searched) at least once.

Let m be the length of $\mathcal{L}$.

Let $p(x)$ be the relative frequency of the search operations that are for element x in the sequence $\mathcal{L}$. Therefore, we have:

- $p(x) \cdot m \geq 1$, and
- $\sum_{x \in P} p(x) = 1$.

Static Optimality

Actual Overall Running Time of a Splay Tree on $\mathcal{L}$

Next, we show that the actual overall cost of a splay tree T on the search sequence $\mathcal{L}$ is bounded by:

$$O(\sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)} + m).$$

Static Optimality

Actual Overall Running Time of a Splay Tree on $\mathcal{L}$

Again, our proof still utilises the potential function

$$\Phi(S_i) = \sum_{x \in T} r(x),$$

but, at this time, the weight for each node x is set as $w(x) = p(x)$ other than being set as $w(x) = 1$.

Think:

Will it invalidate our previous conclusions on the amortized bound for each of the operations, if we use different setting for $w(x)$? Why?

Answer: No, because the amortized analysis is purely conceptual and it does not affect the actual behaviours of a data structure.

Static Optimality

Actual Overall Running Time of a Splay Tree on $\mathcal{L}$

Before we proceed, first observe that:

- in the initial state S_0 of our analysis, the splay tree has n nodes;
- $s(root) = W = \sum_{x \in T} w(x) = \sum_{x \in T} p(x) = 1$;
- $r(x) = \log s(x) \leq \log s(root) = \log W = 0$;
- $\Phi(S_0) = \sum_{x \in T} r^{(0)}(x) \leq \sum_{x \in T} \log W$;
- $\Phi(S_m) = \sum_{x \in T} r^{(1)}(x) = \sum_{x \in T} \log s^{(1)}(x) \geq \sum_{x \in T} \log w(x)$;
- $\Phi(S_0) - \Phi(S_m) \leq \sum_{x \in T} \log \frac{W}{w(x)} = \sum_{x \in T} \log \frac{1}{p(x)}$.

This is for the first time that we meet $\Phi(S_0) \leq 0$ and $\Phi(S_i) \leq 0$ for all integer indexes $i \geq 1$ in this course.

Actual Overall Running Time of a Splay Tree on $\mathcal{L}$

Recall that:

$$\text{overall amortized cost} = \text{overall actual cost} + \Delta\Phi$$

$$amortized(\mathcal{L}) = cost(\mathcal{L}) + \Phi(S_m) - \Phi(S_0)$$

$$\iff \quad cost(\mathcal{L}) = amortized(\mathcal{L}) - (\Phi(S_m) - \Phi(S_0))$$

In the previous lectures, we required that $\Delta\Phi = \Phi(S_m) - \Phi(S_0) \geq 0$ always holds, so as to ensure $amortized(\mathcal{L}) \geq cost(\mathcal{L})$ holds for any operation sequence.

However, if we are only interested in the *asymptotic* overall cost, even though $\Delta\Phi < 0$, as long as $|\Delta\Phi|$ is bounded by some slowly growing function such that

$amortized(\mathcal{L})$ *dominates* $|\Delta\Phi|$ when $m = |\mathcal{L}|$ is sufficiently large,

we still can have: $cost(\mathcal{L}) = \Theta(amortized(\mathcal{L}))$.

Actual Overall Running Time of a Splay Tree on $\mathcal{L}$

As aforementioned, $\Delta\Phi = \Phi(S_m) - \Phi(S_0) \leq -\sum_{x\in T} \log \frac{1}{p(x)}$.

By the Access Lemma and our earlier analysis, we know that the amortized cost of *search*(x) is at most:

$$3\cdot\left(r^{(1)}(x) - r^{(0)}(x)\right)+1 \leq 3\cdot(\log s(root) - \log w(x))+1 = 3\cdot\log \frac{1}{p(x)}+1.$$

Therefore, we have:

$$\begin{aligned}
amortized(\mathcal{L}) &\leq \sum_{x\in T} p(x)\cdot m\cdot(3\cdot\log\frac{1}{p(x)}+1) \\
&= \left(3\cdot\sum_{x\in T} p(x)\cdot m\cdot\log\frac{1}{p(x)}\right) + m\cdot\sum_{x\in T} p(x) \\
&= \left(3\cdot\sum_{x\in T} p(x)\cdot m\cdot\log\frac{1}{p(x)}\right) + m
\end{aligned}$$

Static Optimality

Actual Overall Running Time of a Splay Tree on $\mathcal{L}$

$$
\begin{aligned}
cost(\mathcal{L}) &= amortized(\mathcal{L}) - \Delta\Phi \\
&\leq \left(3 \cdot \sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)}\right) + m - (-\sum_{x \in T} \log \frac{1}{p(x)}) \\
&= \left(\sum_{x \in T} (3 \cdot p(x) \cdot m + 1) \cdot \log \frac{1}{p(x)}\right) + m \\
&= O(\sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)} + m),
\end{aligned}
$$

where the big-O bound follows by the fact that $p(x) \cdot m \geq 1$ for all $x \in T$.

Non-Examinable Extra Reading

Static Optimality

Next, we show that $O(\sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)} + m)$, the overall cost bound, is actually optimal by Information Theory.

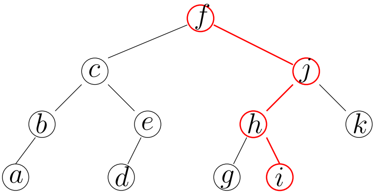
Static Optimality

Tree Path Encoding

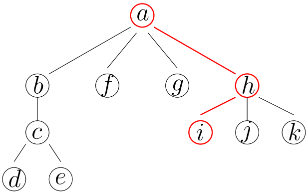
Consider an arbitrary (i.e., not necessarily to be binary) search tree T; for any node u in T, define the branch of u, denoted by $b(u)$, as the number of u's child nodes.

To perform a *search*(x) operation in T, we need to descend a path from *root* to x.

More specifically, starting from $u = root$, if u is a non-leaf node, we select one of u's child nodes to descend the current path at u one step towards the target node x.



example search tree 1



example search tree 2

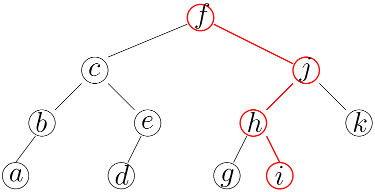
Static Optimality

Tree Path Encoding

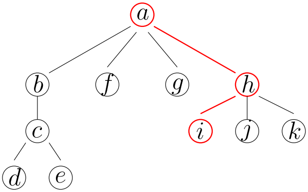
By a standard binary search, the cost for each path descending step is bounded by $\Theta(\lfloor \log b(u) \rfloor + 1)$.

Denote the unique path from *root* to a node u in T by $path(u)$. The cost for $search(x)$ is bounded by $\Theta\left(\sum_{u \in path(x)} (\lfloor \log b(u) \rfloor + 1)\right)$.

In particular, if T is a binary search tree, $b(u) \leq 2$ for all u; the cost for $search(x)$ becomes $\Theta\left(depth(x)\right)$, the depth of x in T.



example search tree 1



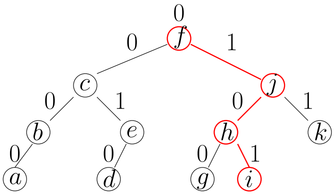
example search tree 2

Static Optimality

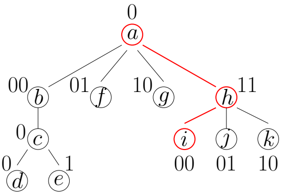
Tree Path Encoding

Essentially, *path*(u) uniquely encodes u, for each node u in T. That is, given a simple path *path*(u) in T starting from *root*, we can uniquely reach to the node u.

Consider *path*(x); for each path descending step from a non-leaf node u, we can use at most $\lfloor \log b(u) \rfloor + 1$ bits to encode the child node of u that is selected in *path*(x).



example search tree 1



example search tree 2

Static Optimality

Tree Path Encoding

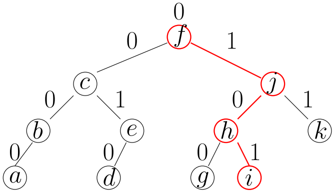
Therefore, $path(x)$ can be encoded as follows:

- initialize the encoding of $path(x)$ as "$0|$", where "$|$" is a special symbol used to separate the encoding of each node in $path(x)$;
- for each path descending step from a node u,
  - append at most $\lfloor \log b(u) \rfloor + 1$ bits for encoding the child node selected in the path;
  - append "$|$" indicating that the end of the child node encoding;
- when reach to the end of $path(x)$, append another special symbol "\$" indicating that the end of the path encoding.

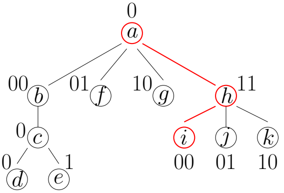
Clearly, the alphabet for encoding $path(x)$ is just $\{0, 1, |, \$\}$, of which each symbol can be represented by 2 bits.

The Static Optimality

Tree Path Encoding



encoding for i: "0|1|0|1|\$"



encoding for i: "0|11|00|\$"

Therefore, the length of the encoding of $path(x)$ is bounded by $O\left(\sum_{u \in path(x)} (\lfloor \log b(u) \rfloor + 1)\right)$ bits, which is exactly the same as the upper bound of the cost for $search(x)$.

Moreover, such an encoding uniquely identifies x.

Static Optimality

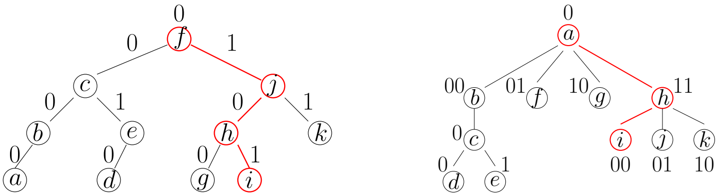
Search Sequence Encoding

Consider the search sequence $\mathcal{L}$; with a search tree T, we can uniquely encode each target node x in the search operations by the encoding of $path(x)$ in T.

For example, for a search sequence $\mathcal{L}$, which searches elements:

$$i, d, e, a,$$

respectively in order.



0|1|0|1|\$0|0|1|0|\$0|0|1|\$0|0|0|0|\$ 0|11|00|\$0|00|0|0|\$0|00|0|1|\$0|\$

Static Optimality

Search Sequence Encoding

- *code-length*$_T(\mathcal{L})$: the overall length of the encoding of $\mathcal{L}$ with T;
- *cost*$_T(\mathcal{L})$: the overall search cost of T for $\mathcal{L}$.

We have:

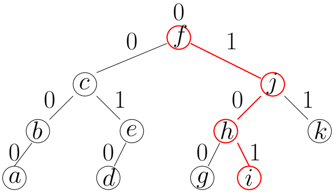
$$m \leq \textit{code-length}_T(\mathcal{L}) = O\left(\textit{cost}_T(\mathcal{L})\right), \tag{3}$$

where the inequality comes from the fact that each target node encoding requires at least one bit.

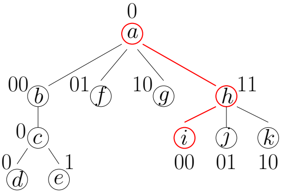
Static Optimality

Search Sequence Encoding

Furthermore, such an encoding of $\mathcal{L}$ is lossless, because $\mathcal{L}$ can be precisely recovered from this encoding with T.



0|1|0|1|\$0|0|1|0|\$0|0|1|\$0|0|0|0|\$



0|11|00|\$0|00|0|0|\$0|00|0|1|\$0|\$

Static Optimality

The Minimum Possible Code Length

By Shannon's source coding theorem, *any* lossless encoding of $\mathcal{L}$ must have at least

$$\sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)}$$

bits; the above quantity is the so-called Shannon's Entropy.

As a result, for any static search tree T, we have:

$$\sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)} \leq \mathit{code\text{-}length}_T(\mathcal{L}) = O\left(\mathit{cost}_T(\mathcal{L})\right). \tag{4}$$

Static Optimality

The Minimum Possible Code Length

Putting (3) and (4) together, we have:

For any static search tree T, the overall cost for $\mathcal{L}$ is at least

$$\Omega(\sum_{x \in T} p(x) \cdot m \cdot \log \frac{1}{p(x)} + m).$$

Therefore, the static optimality of the splay tree follows.