# Randomised Skip Lists

Max W. Haslbeck, Manuel Eberl

March 19, 2025

**Abstract**

Skip lists are sorted linked lists enhanced with shortcuts and are an alternative to binary search trees [2]. A skip lists consists of multiple levels of sorted linked lists where a list on level $n$ is a subsequence of the list on level $n-1$. In the ideal case, elements are *skipped* in such a way that a lookup in a skip lists takes $\mathcal{O}(\log n)$ time. In a randomised skip list the skipped elements are choosen randomly.

This entry contains formalized proofs of the textbook results about the expected height and the expected length of a search path in a randomised skip list [1].

## Contents

# 1 Indexed products of PMFs

**theory** *Pi-pmf*
  **imports** *HOL−Probability.Probability*
**begin**

Conflicting notation from *HOL−Analysis.Infinite-Sum*

**no-notation** *Infinite-Sum.abs-summable-on* (**infixr** ‹*abs′-summable′-on*› *46*)

## 1.1 Definition

In analogy to $Pi_M$, we define an indexed product of PMFs. In the literature, this is typically called taking a vector of independent random variables. Note that the components do not have to be identically distributed.

The operation takes an explicit index set $A$ and a function $f$ that maps each element from $A$ to a PMF and defines the product measure $\bigotimes_{i \in A} f(i)$ , which is represented as a $('a \Rightarrow 'b)$ *pmf*.

Note that unlike $Pi_M$, this only works for *finite* index sets. It could be extended to countable sets and beyond, but the construction becomes somewhat more involved.

**definition** *Pi-pmf* :: *′a set* ⇒ *′b* ⇒ (*′a* ⇒ *′b pmf*) ⇒ (*′a* ⇒ *′b*) *pmf* **where**
  *Pi-pmf A dflt p* =
    *embed-pmf* (λ*f*. *if* (∀ *x*. *x* ∉ *A* ⟶ *f x* = *dflt*) *then* ∏ *x*∈*A*. *pmf* (*p x*) (*f x*) *else 0*)

A technical subtlety that needs to be addressed is this: Intuitively, the functions in the support of a product distribution have domain $A$. However, since HOL is a total logic, these functions must still return *some* value for inputs outside $A$. The product measure $Pi_M$ simply lets these functions return *undefined* in these cases. We chose a different solution here, which is to supply a default value *dflt* that is returned in these cases.

As one possible application, one could model the result of $n$ different independent coin tosses as *Pi-pmf.Pi-pmf* {*0*..<*n*} *False* (λ-. *bernoulli-pmf* (*1* / *2*)). This returns a function of type *nat* ⇒ *bool* that maps every natural number below $n$ to the result of the corresponding coin toss, and every other natural number to *False*.

**lemma** *pmf-Pi*:
  **assumes** *A*: *finite A*
  **shows** *pmf* (*Pi-pmf A dflt p*) *f* =
      (*if* (∀ *x*. *x* ∉ *A* ⟶ *f x* = *dflt*) *then* ∏ *x*∈*A*. *pmf* (*p x*) (*f x*) *else 0*)
  ⟨*proof*⟩

**lemma** *pmf-Pi′*:
  **assumes** *finite A* ⋀*x*. *x* ∉ *A* ⟹ *f x* = *dflt*
  **shows** *pmf* (*Pi-pmf A dflt p*) *f* = (∏ *x*∈*A*. *pmf* (*p x*) (*f x*))

⟨*proof*⟩

**lemma** *pmf-Pi-outside*:
**assumes** $finite\ A\ \exists x.\ x \notin A \wedge f\ x \neq dflt$
**shows** $pmf\ (Pi\text{-}pmf\ A\ dflt\ p)\ f = 0$
⟨*proof*⟩

**lemma** *pmf-Pi-empty* [*simp*]: $Pi\text{-}pmf\ \{\}\ dflt\ p = return\text{-}pmf\ (\lambda\text{-}.\ dflt)$
⟨*proof*⟩

**lemma** *set-Pi-pmf-subset*: $finite\ A \Longrightarrow set\text{-}pmf\ (Pi\text{-}pmf\ A\ dflt\ p) \subseteq \{f.\ \forall x.\ x \notin A \longrightarrow f\ x = dflt\}$
⟨*proof*⟩

**lemma** *Pi-pmf-cong* [*cong*]:
**assumes** $A = A'\ dflt = dflt'\ \bigwedge x.\ x \in A \Longrightarrow f\ x = f'\ x$
**shows** $Pi\text{-}pmf\ A\ dflt\ f = Pi\text{-}pmf\ A'\ dflt'\ f'$
⟨*proof*⟩

## 1.2 Dependent product sets with a default

The following describes a dependent product of sets where the functions are required to return the default value *dflt* outside their domain, in analogy to $Pi_E$, which uses *undefined*.

**definition** *PiE-dflt*
**where** $PiE\text{-}dflt\ A\ dflt\ B = \{f.\ \forall x.\ (x \in A \longrightarrow f\ x \in B\ x) \wedge (x \notin A \longrightarrow f\ x = dflt)\}$

**lemma** *restrict-PiE-dflt*: $(\lambda h.\ restrict\ h\ A)$ ‘ $PiE\text{-}dflt\ A\ dflt\ B = PiE\ A\ B$
⟨*proof*⟩

**lemma** *dflt-image-PiE*: $(\lambda h\ x.\ if\ x \in A\ then\ h\ x\ else\ dflt)$ ‘ $PiE\ A\ B = PiE\text{-}dflt\ A\ dflt\ B$
(**is** *?f* ‘ *?X* = *?Y*)
⟨*proof*⟩

**lemma** *finite-PiE-dflt* [*intro*]:
**assumes** $finite\ A\ \bigwedge x.\ x \in A \Longrightarrow finite\ (B\ x)$
**shows** $finite\ (PiE\text{-}dflt\ A\ d\ B)$
⟨*proof*⟩

**lemma** *card-PiE-dflt*:
**assumes** $finite\ A\ \bigwedge x.\ x \in A \Longrightarrow finite\ (B\ x)$
**shows** $card\ (PiE\text{-}dflt\ A\ d\ B) = (\prod x{\in}A.\ card\ (B\ x))$
⟨*proof*⟩

**lemma** *PiE-dflt-empty-iff* [*simp*]: $PiE\text{-}dflt\ A\ dflt\ B = \{\} \longleftrightarrow (\exists x{\in}A.\ B\ x = \{\})$
⟨*proof*⟩

The probability of an independent combination of events is precisely the product of the probabilities of each individual event.

**lemma** *measure-Pi-pmf-PiE-dflt*:
**assumes** [*simp*]: *finite A*
**shows** *measure-pmf.prob* (*Pi-pmf A dflt p*) (*PiE-dflt A dflt B*) =
($\prod$ *x*∈*A*. *measure-pmf.prob* (*p x*) (*B x*))
⟨*proof*⟩

**lemma** *set-Pi-pmf-subset′*:
**assumes** *finite A*
**shows** *set-pmf* (*Pi-pmf A dflt p*) ⊆ *PiE-dflt A dflt* (*set-pmf* ∘ *p*)
⟨*proof*⟩

**lemma** *Pi-pmf-return-pmf* [*simp*]:
**assumes** *finite A*
**shows** *Pi-pmf A dflt* (λ*x*. *return-pmf* (*f x*)) = *return-pmf* (λ*x*. *if x* ∈ *A then f x else dflt*)
⟨*proof*⟩

**lemma** *Pi-pmf-return-pmf′* [*simp*]:
**assumes** *finite A*
**shows** *Pi-pmf A dflt* (λ-. *return-pmf dflt*) = *return-pmf* (λ-. *dflt*)
⟨*proof*⟩

**lemma** *measure-Pi-pmf-Pi*:
**fixes** *t*::*nat*
**assumes** [*simp*]: *finite A*
**shows** *measure-pmf.prob* (*Pi-pmf A dflt p*) (*Pi A B*) =
($\prod$ *x*∈*A*. *measure-pmf.prob* (*p x*) (*B x*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

## 1.3 Common PMF operations on products

*Pi-pmf.Pi-pmf* distributes over the ‘bind’ operation in the Giry monad:

**lemma** *Pi-pmf-bind*:
**assumes** *finite A*
**shows** *Pi-pmf A d* (λ*x*. *bind-pmf* (*p x*) (*q x*)) =
*do* {*f* ← *Pi-pmf A d′ p*; *Pi-pmf A d* (λ*x*. *q x* (*f x*))} (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Analogously any componentwise mapping can be pulled outside the product:

**lemma** *Pi-pmf-map*:
**assumes** [*simp*]: *finite A* **and** *f dflt* = *dflt′*
**shows** *Pi-pmf A dflt′* (λ*x*. *map-pmf f* (*g x*)) = *map-pmf* (λ*h*. *f* ∘ *h*) (*Pi-pmf A dflt g*)
⟨*proof*⟩

We can exchange the default value in a product of PMFs like this:

**lemma** *Pi-pmf-default-swap*:
**assumes** *finite A*
**shows** *map-pmf* $(\lambda f\ x.\ if\ x \in A\ then\ f\ x\ else\ dflt')$ $(Pi\text{-}pmf\ A\ dflt\ p) =$
*Pi-pmf A dflt′ p* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

The following rule allows reindexing the product:

**lemma** *Pi-pmf-bij-betw*:
**assumes** *finite A bij-betw h A B* $\bigwedge x.\ x \notin A \Longrightarrow h\ x \notin B$
**shows** *Pi-pmf A dflt* $(\lambda\text{-}.\ f)$ = *map-pmf* $(\lambda g.\ g \circ h)$ (*Pi-pmf B dflt* $(\lambda\text{-}.\ f)$)
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

A product of uniform random choices is again a uniform distribution.

**lemma** *Pi-pmf-of-set*:
**assumes** *finite A* $\bigwedge x.\ x \in A \Longrightarrow$ *finite* $(B\ x)$ $\bigwedge x.\ x \in A \Longrightarrow B\ x \neq \{\}$
**shows** *Pi-pmf A d* $(\lambda x.$ *pmf-of-set* $(B\ x))$ = *pmf-of-set* (*PiE-dflt A d B*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

## 1.4 Merging and splitting PMF products

The following lemma shows that we can add a single PMF to a product:

**lemma** *Pi-pmf-insert*:
**assumes** *finite A* $x \notin A$
**shows** *Pi-pmf* (*insert x A*) *dflt p* = *map-pmf* $(\lambda(y,f).\ f(x{:=}y))$ (*pair-pmf* (*p x*) (*Pi-pmf A dflt p*))
⟨*proof*⟩

**lemma** *Pi-pmf-insert′*:
**assumes** *finite A* $x \notin A$
**shows** *Pi-pmf* (*insert x A*) *dflt p* =
*do* $\{y \leftarrow p\ x;\ f \leftarrow$ *Pi-pmf A dflt p*; *return-pmf* $(f(x := y))\}$
⟨*proof*⟩

**lemma** *Pi-pmf-singleton*:
*Pi-pmf* $\{x\}$ *dflt p* = *map-pmf* $(\lambda a\ b.\ if\ b = x\ then\ a\ else\ dflt)$ $(p\ x)$
⟨*proof*⟩

Projecting a product of PMFs onto a component yields the expected result:

**lemma** *Pi-pmf-component*:
**assumes** *finite A*
**shows** *map-pmf* $(\lambda f.\ f\ x)$ (*Pi-pmf A dflt p*) = (*if* $x \in A$ *then p x else return-pmf dflt*)
⟨*proof*⟩

We can take merge two PMF products on disjoint sets like this:

**lemma** *Pi-pmf-union*:

**assumes** *finite A finite B* $A \cap B = \{\}$
**shows** *Pi-pmf* $(A \cup B)$ *dflt p* =
*map-pmf* ($\lambda(f,g)$ *x. if* $x \in A$ *then f x else g x*)
(*pair-pmf* (*Pi-pmf A dflt p*) (*Pi-pmf B dflt p*)) (**is** - = *map-pmf* (*?h A*) (*?q A*))
⟨*proof*⟩

We can also project a product to a subset of the indices by mapping all the other indices to the default value:

**lemma** *Pi-pmf-subset*:
**assumes** *finite A* $A' \subseteq A$
**shows** *Pi-pmf* $A'$ *dflt p* = *map-pmf* ($\lambda f$ *x. if* $x \in A'$ *then f x else dflt*) (*Pi-pmf A dflt p*)
⟨*proof*⟩

**lemma** *Pi-pmf-subset'*:
**fixes** *f* :: *'a* ⇒ *'b pmf*
**assumes** *finite A* $B \subseteq A$ $\bigwedge x.\ x \in A - B \Longrightarrow f\ x = $ *return-pmf dflt*
**shows** *Pi-pmf A dflt f* = *Pi-pmf B dflt f*
⟨*proof*⟩

**lemma** *Pi-pmf-if-set*:
**assumes** *finite A*
**shows** *Pi-pmf A dflt* ($\lambda x$. *if b x then f x else return-pmf dflt*) =
*Pi-pmf* $\{x \in A.\ b\ x\}$ *dflt f*
⟨*proof*⟩

**lemma** *Pi-pmf-if-set'*:
**assumes** *finite A*
**shows** *Pi-pmf A dflt* ($\lambda x$. *if b x then return-pmf dflt else f x*) =
*Pi-pmf* $\{x \in A.\ \neg b\ x\}$ *dflt f*
⟨*proof*⟩

Lastly, we can delete a single component from a product:

**lemma** *Pi-pmf-remove*:
**assumes** *finite A*
**shows** *Pi-pmf* $(A - \{x\})$ *dflt p* = *map-pmf* ($\lambda f.\ f(x := dflt)$) (*Pi-pmf A dflt p*)
⟨*proof*⟩

## 1.5 Applications

Choosing a subset of a set uniformly at random is equivalent to tossing a fair coin independently for each element and collecting all the elements that came up heads.

**lemma** *pmf-of-set-Pow-conv-bernoulli*:
**assumes** *finite* (*A* :: *'a set*)

**shows** *map-pmf* (λ*b*. {*x*∈*A*. *b x*}) (*Pi-pmf A P* (λ-. *bernoulli-pmf* (*1/2*))) = *pmf-of-set* (*Pow A*)
⟨*proof*⟩

A binomial distribution can be seen as the number of successes in $n$ independent coin tosses.

**lemma** *binomial-pmf-altdef′*:
**fixes** *A* :: *′a set*
**assumes** *finite A* **and** *card A* = *n* **and** *p*: *p* ∈ {*0..1*}
**shows** *binomial-pmf n p* =
*map-pmf* (λ*f*. *card* {*x*∈*A*. *f x*}) (*Pi-pmf A dflt* (λ-. *bernoulli-pmf p*)) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

**end**

# 2 Auxiliary material

**theory** *Misc*
**imports** *HOL−Analysis.Analysis*
**begin**

Based on *sorted-list-of-set* and *the-inv-into* we construct a bijection between a finite set A of type 'a::linorder and a set of natural numbers {..<*card A*}

**lemma** *bij-betw-mono-on-the-inv-into*:
**fixes** *A*::*′a*::*linorder set* **and** *B*::*′b*::*linorder set*
**assumes** *b*: *bij-betw f A B* **and** *m*: *mono-on A f*
**shows** *mono-on B* (*the-inv-into A f*)
⟨*proof*⟩

**lemma** *rev-removeAll-removeAll-rev*: *rev* (*removeAll x xs*) = *removeAll x* (*rev xs*)
⟨*proof*⟩

**lemma** *sorted-list-of-set-Min-Cons*:
**assumes** *finite A A* ≠ {}
**shows** *sorted-list-of-set A* = *Min A* # *sorted-list-of-set* (*A* − {*Min A*})
⟨*proof*⟩

**lemma** *sorted-list-of-set-filter*:
**assumes** *finite A*
**shows** *sorted-list-of-set* ({*x*∈*A*. *P x*}) = *filter P* (*sorted-list-of-set A*)
⟨*proof*⟩

**lemma** *sorted-list-of-set-Max-snoc*:
**assumes** *finite A A* ≠ {}
**shows** *sorted-list-of-set A* = *sorted-list-of-set* (*A* − {*Max A*}) @ [*Max A*]
⟨*proof*⟩

**lemma** *sorted-list-of-set-image*:
**assumes** *mono-on A g inj-on g A*
**shows** (*sorted-list-of-set* (*g* ‘ *A*)) = *map g* (*sorted-list-of-set A*)
⟨*proof*⟩

**lemma** *sorted-list-of-set-length*: *length* (*sorted-list-of-set A*) = *card A*
⟨*proof*⟩

**lemma** *sorted-list-of-set-bij-betw*:
**assumes** *finite A*
**shows** *bij-betw* (λ*n. sorted-list-of-set A* ! *n*) {..<*card A*} *A*
⟨*proof*⟩

**lemma** *nth-mono-on*:
**assumes** *sorted xs distinct xs set xs* = *A*
**shows** *mono-on* {..<*card A*} (λ*n. xs* ! *n*)
⟨*proof*⟩

**lemma** *sorted-list-of-set-mono-on*:
*finite A* ⟹ *mono-on* {..<*card A*} (λ*n. sorted-list-of-set A* ! *n*)
⟨*proof*⟩

**definition** *bij-mono-map-set-to-nat* :: *'a*::*linorder set* ⇒ *'a* ⇒ *nat* **where**
*bij-mono-map-set-to-nat A* =
(λ*x. if x* ∈ *A then the-inv-into* {..<*card A*} ((!) (*sorted-list-of-set A*)) *x*
*else card A*)

**lemma** *bij-mono-map-set-to-nat*:
**assumes** *finite A*
**shows** *bij-betw* (*bij-mono-map-set-to-nat A*) *A* {..<*card A*}
*mono-on A* (*bij-mono-map-set-to-nat A*)
(*bij-mono-map-set-to-nat A*) ‘ *A* = {..<*card A*}
⟨*proof*⟩

**end**

# 3 Theorems about the Geometric Distribution

**theory** *Geometric-PMF*
**imports**
*HOL−Probability.Probability*
*Pi-pmf*
*Monad-Normalisation.Monad-Normalisation*
**begin**

**lemma** *nn-integral-geometric-pmf*:
**assumes** *p* ∈ {*0*<..*1*}
**shows** *nn-integral* (*geometric-pmf p*) *real* = (*1* − *p*) / *p*
⟨*proof*⟩

**lemma** *geometric-pmf-prob-atMost*:
**assumes** $p \in \{0<..1\}$
**shows** *measure-pmf.prob* (*geometric-pmf p*) {..*n*} = (*1* − (*1* − *p*)^(*n* + *1*))
⟨*proof*⟩

**lemma** *geometric-pmf-prob-lessThan*:
**assumes** $p \in \{0<..1\}$
**shows** *measure-pmf.prob* (*geometric-pmf p*) {..<*n*} = *1* − (*1* − *p*) ^ *n*
⟨*proof*⟩

**lemma** *geometric-pmf-prob-greaterThan*:
**assumes** $p \in \{0<..1\}$
**shows** *measure-pmf.prob* (*geometric-pmf p*) {*n*<..} = (*1* − *p*)^(*n* + *1*)
⟨*proof*⟩

**lemma** *geometric-pmf-prob-atLeast*:
**assumes** $p \in \{0<..1\}$
**shows** *measure-pmf.prob* (*geometric-pmf p*) {*n*..} = (*1* − *p*)^*n*
⟨*proof*⟩

**lemma** *bernoulli-pmf-of-set′*:
**assumes** *finite A*
**shows** *map-pmf* (λ*b*. {*x* ∈ *A*. ¬ *b x*}) (*Pi-pmf A P* (λ-. *bernoulli-pmf* (*1*/*2*)))
= *pmf-of-set* (*Pow A*)
⟨*proof*⟩

**lemma** *Pi-pmf-pmf-of-set-Suc*:
**assumes** *finite A*
**shows** *Pi-pmf A 0* (λ-. *geometric-pmf* (*1*/*2*)) =
*do* {
*B* ← *pmf-of-set* (*Pow A*);
*Pi-pmf B 0* (λ-. *map-pmf Suc* (*geometric-pmf* (*1*/*2*))) }
⟨*proof*⟩

**lemma** *Pi-pmf-pmf-of-set-Suc′*:
**assumes** *finite A*
**shows** *Pi-pmf A 0* (λ-. *geometric-pmf* (*1*/*2*)) =
*do* {
*B* ← *pmf-of-set* (*Pow A*);
*Pi-pmf B 0* (λ-. *map-pmf Suc* (*geometric-pmf* (*1*/*2*))) }
⟨*proof*⟩

**lemma** *binomial-pmf-altdef′*:
**fixes** *A* :: *′a set*
**assumes** *finite A* **and** *card A* = *n* **and** *p*: $p \in \{0..1\}$
**shows** *binomial-pmf n p* =
*map-pmf* (λ*f*. *card* {*x*∈*A*. *f x*}) (*Pi-pmf A dflt* (λ-. *bernoulli-pmf p*)) (**is**
*?lhs* = *?rhs*)

⟨*proof*⟩

**lemma** *bernoulli-pmf-Not*:
  **assumes** $p \in \{0..1\}$
  **shows** *bernoulli-pmf p* = *map-pmf Not* (*bernoulli-pmf* (*1* − *p*))
⟨*proof*⟩

**lemma** *binomial-pmf-altdef''*:
  **assumes** *p*: $p \in \{0..1\}$
  **shows**   *binomial-pmf n p* =
      *map-pmf* (λ*f*. *card* {*x*. *x* < *n* ∧ *f x*}) (*Pi-pmf* {..<*n*} *dflt* (λ-. *bernoulli-pmf*
*p*))
  ⟨*proof*⟩

**context includes** *monad-normalisation*
**begin**

**lemma** *Pi-pmf-geometric-filter*:
  **assumes** *finite A* $p \in \{0<..1\}$
  **shows** *Pi-pmf A 0* (λ-. *geometric-pmf p*) =
     *do* {
      *fb* ← *Pi-pmf A dflt* (λ-. *bernoulli-pmf p*);
      *Pi-pmf* {*x* ∈ *A*. ¬ *fb x*} *0* (λ-. *map-pmf Suc* (*geometric-pmf p*)) }
⟨*proof*⟩

**lemma** *Pi-pmf-geometric-filter'*:
  **assumes** *finite A* $p \in \{0<..1\}$
  **shows** *Pi-pmf A 0* (λ-. *geometric-pmf p*) =
     *do* {
      *fb* ← *Pi-pmf A dflt* (λ-. *bernoulli-pmf* (*1* − *p*));
      *Pi-pmf* {*x* ∈ *A*. *fb x*} *0* (λ-. *map-pmf Suc* (*geometric-pmf p*)) }
  ⟨*proof*⟩

**end**

**end**

# 4 Randomized Skip Lists

**theory** *Skip-List*
  **imports** *Geometric-PMF*
      *Misc*
      *Monad-Normalisation.Monad-Normalisation*
**begin**

Conflicting notation from *HOL−Analysis.Infinite-Sum*

**no-notation** *Infinite-Sum.abs-summable-on* (**infixr** ‹*abs'-summable'-on*› *46*)

## 4.1 Preliminaries

**lemma** *bind-pmf-if'*: $(do\ \{c \leftarrow C;$
$ab \leftarrow (if\ c\ then\ A\ else\ B);$
$D\ ab\}::'a\ pmf) =$
$do\ \{c \leftarrow C;$
$(if\ c\ then\ (A \gg= D)\ else\ (B \gg= D))\}$
⟨*proof*⟩

**abbreviation** (*input*) $Max_0$ **where** $Max_0 \equiv (\lambda A.\ Max\ (A \cup \{0\}))$

## 4.2 Definition of a Randomised Skip List

Given a set A we assign a geometric random variable (counting the number of failed Bernoulli trials before the first success) to every element in A. That means an arbitrary element of A is on level n with probability $(1 - p)^n p$. We define he height of the skip list as the maximum assigned level. So a skip list with only one level has height 0 but the calculation of the expected height is cleaner this way.

**locale** *random-skip-list* =
**fixes** $p$::*real*
**begin**

**definition** $q$ **where** $q = 1 - p$

**definition** $SL :: ('a::linorder)\ set \Rightarrow ('a \Rightarrow nat)\ pmf$ **where** $SL\ A = Pi\text{-}pmf\ A\ 0\ (\lambda\text{-}.\ geometric\text{-}pmf\ p)$
**definition** $SL_N :: nat \Rightarrow (nat \Rightarrow nat)\ pmf$ **where** $SL_N\ n = SL\ \{..<n\}$

## 4.3 Height of Skip List

**definition** $H$ **where** $H\ A = map\text{-}pmf\ (\lambda f.\ Max_0\ (f\ `\ A))\ (SL\ A)$
**definition** $H_N :: nat \Rightarrow nat\ pmf$ **where** $H_N\ n = H\ \{..<n\}$

**context includes** *monad-normalisation*
**begin**

The height of a skip list is independent of the values in a set A. For simplicity we can therefore work on the skip list over the set $\{..<card\ A\}$

**lemma**
**assumes** *finite A*
**shows** $H\ A = H_N\ (card\ A)$
⟨*proof*⟩

The cumulative distribution function (CDF) of the height is the CDF of the geometric PMF to the power of n

**lemma** *prob-Max-IID-geometric-atMost*:
**assumes** $p \in \{0..1\}$

**shows** *measure-pmf.prob* $(H_N\ n)$ *{..i}*
= (*measure-pmf.prob* (*geometric-pmf p*) *{..i}*) ^ *n* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

**lemma** *prob-Max-IID-geometric-greaterThan*:
**assumes** $p \in \{0<..1\}$
**shows** *measure-pmf.prob* $(H_N\ n)$ *{i<..}* =
*1* − (*1* − *q* ^ (*i* + *1*)) ^ *n*
⟨*proof*⟩

**end**
**end**

An alternative definition of the expected value of a non-negative random variable [1]

**lemma** *expectation-prob-atLeast*:
**assumes** (λ*i. measure-pmf.prob N {i..}*) *abs-summable-on {1..}*
**shows** *measure-pmf.expectation N real* = *infsetsum* (λ*i. measure-pmf.prob N {i..}*) *{1..}*
*integrable N real*
⟨*proof*⟩

The expected height of a skip list has no closed-form expression but we can approximate it. We start by showing how we can calculate an infinite sum over the natural numbers with an integral over the positive reals and the floor function.

**lemma** *infsetsum-set-nn-integral-reals*:
**assumes** *f abs-summable-on UNIV* $\bigwedge n.\ f\ n \geq 0$
**shows** *infsetsum f UNIV* = *set-nn-integral lborel* {*0*::*real..*} (λ*x. f* (*nat* (*floor x*)))
⟨*proof*⟩

**lemma** *nn-integral-nats-reals*:
**shows** $(\int^+ i.\ ennreal\ (f\ i)\ \partial count\text{-}space\ UNIV) = (\int^+ x \in \{0::real..\}.\ ennreal\ (f\ (nat\ \lfloor x \rfloor))\partial lborel)$
⟨*proof*⟩

**lemma** *nn-integral-floor-less-eq*:
**assumes** $\bigwedge x\ y.\ x \leq y \Longrightarrow f\ y \leq f\ x$
**shows** $(\int^+ x \in \{0::real..\}.\ ennreal\ (f\ x)\partial lborel) \leq (\int^+ x \in \{0::real..\}.\ ennreal\ (f\ (nat\ \lfloor x \rfloor))\partial lborel)$
⟨*proof*⟩

**lemma** *nn-integral-finite-imp-abs-sumable-on*:
**fixes** *f* :: *'a* ⇒ *'b*::{*banach*, *second-countable-topology*}
**assumes** *nn-integral* (*count-space A*) (λ*x. norm* (*f x*)) < ∞

[1] https://en.wikipedia.org/w/index.php?title=Expected_value&oldid=881384346#Formula_for_non-negative_random_variables

**shows** *f abs-summable-on A*
⟨*proof*⟩

**lemma** *nn-integral-finite-imp-abs-sumable-on′*:
**assumes** *nn-integral* (*count-space A*) (λ*x*. *ennreal* (*f x*)) < ∞ ⋀*x*. *f x* ≥ *0*
**shows** *f abs-summable-on A*
⟨*proof*⟩

We now show that $\int_0^\infty 1 - (1 - q^x)^n \, dx = \frac{-H_n}{\ln q}$ if $0 < q < 1$.

**lemma** *harm-integral-x-raised-n*:
*set-integrable lborel* {*0*::*real*..*1*} (λ*x*. (∑ *i*∈{..<*n*}. *x* ^ *i*)) (**is** *?thesis1*)
*LBINT x* = *0*..*1*. (∑ *i*∈{..<*n*}. *x* ^ *i*) = *harm n* (**is** *?thesis2*)
⟨*proof*⟩

**lemma** *harm-integral-0-1-fraction*:
*set-integrable lborel* {*0*::*real*..*1*} (λ*x*. (*1* − *x* ^ *n*) / (*1* − *x*))
(*LBINT x* = *0*..*1*. ((*1* − *x* ^ *n*) / (*1* − *x*))) = *harm n*
⟨*proof*⟩

**lemma** *one-minus-one-minus-q-x-n-integral*:
**assumes** *q* ∈ {*0*<..<*1*}
**shows** *set-integrable lborel* (*einterval 0* ∞) (λ*x*. (*1* − (*1* − *q powr x*) ^ *n*))
(*LBINT x*=*0*..∞. *1* − (*1* − *q powr x*) ^ *n*) = − *harm n* / *ln q*
⟨*proof*⟩

**lemma** *one-minus-one-minus-q-x-n-nn-integral*:
**fixes** *q*::*real*
**assumes** *q* ∈ {*0*<..<*1*}
**shows** *set-nn-integral lborel* {*0*..} (λ*x*. (*1* − (*1* − *q powr x*) ^ *n*)) =
*LBINT x*=*0*..∞. *1* − (*1* − *q powr x*) ^ *n*
⟨*proof*⟩

We can now derive bounds for the expected height.

**context** *random-skip-list*
**begin**

**definition** $EH_N$ **where** $EH_N$ *n* = *measure-pmf*.*expectation* ($H_N$ *n*) *real*

**lemma** $EH_N$*-bounds′*:
**fixes** *n*::*nat*
**assumes** *p* ∈ {*0*<..<*1*} *0* < *n*
**shows** − *harm n* / *ln q* − *1* ≤ $EH_N$ *n*
$EH_N$ *n* ≤ − *harm n* / *ln q*
*integrable* ($H_N$ *n*) *real*
⟨*proof*⟩

**theorem** $EH_N$*-bounds*:
**fixes** *n*::*nat*
**assumes** *p* ∈ {*0*<..<*1*}

**shows**

$- \mathit{harm}\ n\ /\ \ln q - 1 \le EH_N\ n$

$EH_N\ n \le - \mathit{harm}\ n\ /\ \ln q$

*integrable* $(H_N\ n)$ *real*

⟨*proof*⟩

**end**

## 4.4 Expected Length of Search Path

Let $A$ and $f$ where f is an abstract description of a skip list (assign each value its maximum level). steps A f s u l starts on the rightmost element on level s in the skip lists. If possible it moves up, if not it moves to the left. For every step up it adds cost u and for every step to the left it adds cost l. steps A f 0 1 1 therefore walks from the bottom right corner of a skip list to the top left corner of a skip list and counts all steps.

```
function steps :: 'a :: linorder set ⇒ ('a ⇒ nat) ⇒ nat ⇒ nat ⇒ nat ⇒ nat
where
  steps A f l up left = (if A = {} ∨ infinite A
           then 0
           else (let m = Max A in (if f m < l then      steps (A − {m}) f l up left
                                   else (if f m > l then up + steps A f (l + 1) up left
                                   else                  left + steps (A − {m}) f l up left))))
  ⟨proof⟩
termination
⟨proof⟩
```

**declare** *steps.simps*[*simp del*]

lsteps is similar to steps but is using lists instead of sets. This makes the proofs where we use induction easier.

```
function lsteps :: 'a list ⇒ ('a ⇒ nat) ⇒ nat ⇒ nat ⇒ nat ⇒ nat where
  lsteps [] f l up left = 0 |
  lsteps (x#xs) f l up left = (if      f x < l then lsteps xs f l up left
                               else (if f x > l then up + lsteps (x#xs) f (l + 1) up left
                               else                    left + lsteps xs f l up left))
  ⟨proof⟩
termination
⟨proof⟩
```

**declare** *lsteps.simps*(*2*)[*simp del*]

**lemma** *steps-empty* [*simp*]: *steps* {} *f l up left* = *0*
⟨*proof*⟩

**lemma** *steps-lsteps*: *steps A f l u v* = *lsteps* (*rev* (*sorted-list-of-set A*)) *f l u v*
⟨*proof*⟩

**lemma** *lsteps-comp-map*: $lsteps\ zs\ (f \circ g)\ l\ u\ v = lsteps\ (map\ g\ zs)\ f\ l\ u\ v$
⟨*proof*⟩

**lemma** *steps-image*:
**assumes** $finite\ A\ mono\text{-}on\ A\ g\ inj\text{-}on\ g\ A$
**shows** $steps\ A\ (f \circ g)\ l\ u\ v = steps\ (g\ `\ A)\ f\ l\ u\ v$
⟨*proof*⟩

**lemma** *lsteps-cong*:
**assumes** $ys = xs\ \bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x\ l = l'$
**shows** $lsteps\ xs\ f\ l\ u\ v = lsteps\ ys\ g\ l'\ u\ v$
⟨*proof*⟩

**lemma** *steps-cong*:
**assumes** $A = B\ \bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x\ l = l'$
**shows** $steps\ A\ f\ l\ u\ v = steps\ B\ g\ l'\ u\ v$
⟨*proof*⟩

**lemma** *lsteps-f-add′*:
**shows** $lsteps\ xs\ f\ l\ u\ v = lsteps\ xs\ (\lambda x.\ f\ x + m)\ (l + m)\ u\ v$
⟨*proof*⟩

**lemma** *steps-f-add′*:
**shows** $steps\ A\ f\ l\ u\ v = steps\ A\ (\lambda x.\ f\ x + m)\ (l + m)\ u\ v$
⟨*proof*⟩

**lemma** *lsteps-smaller-set*:
**assumes** $m \le l$
**shows** $lsteps\ xs\ f\ l\ u\ v = lsteps\ [x \leftarrow xs.\ m \le f\ x]\ f\ l\ u\ v$
⟨*proof*⟩

**lemma** *steps-smaller-set*:
**assumes** $finite\ A\ m \le l$
**shows** $steps\ A\ f\ l\ u\ v = steps\ \{x \in A.\ f\ x \ge m\}\ f\ l\ u\ v$
⟨*proof*⟩

**lemma** *lsteps-level-greater-fun-image*:
**assumes** $\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x < l$
**shows** $lsteps\ xs\ f\ l\ u\ v = 0$
⟨*proof*⟩

**lemma** *lsteps-smaller-card-Max-fun′*:
**assumes** $\exists\, x \in set\ xs.\ l \le f\ x$
**shows** $lsteps\ xs\ f\ l\ u\ v + l * u \le v * length\ xs + u * Max\ ((f\ `\ (set\ xs)) \cup \{0\})$
⟨*proof*⟩

**lemma** *steps-smaller-card-Max-fun′*:
**assumes** $finite\ A\ \exists\, x \in A.\ l \le f\ x$
**shows** $steps\ A\ f\ l\ up\ left + l * up \le left * card\ A + up * Max_0\ (f\ `\ A)$

⟨*proof*⟩

**lemma** *lsteps-height*:
  **assumes** $\exists\, x \in set\ xs.\ l \leq f\ x$
  **shows** $lsteps\ xs\ f\ l\ up\ 0 + up * l = up * Max_0\ (f\ `\ (set\ xs))$
  ⟨*proof*⟩

**lemma** *steps-height*:
  **assumes** *finite A*
  **shows** $steps\ A\ f\ 0\ up\ 0 = up * Max_0\ (f\ `\ A)$
⟨*proof*⟩

**context** *random-skip-list*
**begin**

We can now define the pmf describing the length of the search path in a skip list. Like the height it only depends on the number of elements in the skip list's underlying set.

**definition** $R$ **where** $R\ A\ u\ l = map\text{-}pmf\ (\lambda f.\ steps\ A\ f\ 0\ u\ l)\ (SL\ A)$
**definition** $R_N :: nat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ pmf$ **where** $R_N\ n\ u\ l = R\ \{..<n\}\ u\ l$

**lemma** $R_N$*-alt-def*: $R_N\ n\ u\ l = map\text{-}pmf\ (\lambda f.\ steps\ \{..<n\}\ f\ 0\ u\ l)\ (SL_N\ n)$
  ⟨*proof*⟩

**context includes** *monad-normalisation*
**begin**

**lemma** $R$-$R_N$:
  **assumes** *finite A* $p \in \{0..1\}$
  **shows** $R\ A\ u\ l = R_N\ (card\ A)\ u\ l$
⟨*proof*⟩

$R_N$ fulfills a recurrence relation. If we move up or to the left the "remaining" length of the search path is again a slightly different probability distribution over the length.

**lemma** $R_N$*-recurrence*:
  **assumes** $0 < n$ $p \in \{0<..1\}$
  **shows** $R_N\ n\ u\ l =$
```
do {
  b ← bernoulli-pmf p;
  if b then                — leftwards
    map-pmf (λn. n + l) (R_N (n − 1) u l)
  else do {                — upwards
    m ← binomial-pmf (n − 1) (1 − p);
    map-pmf (λn. n + u) (R_N (m + 1) u l)
  }
}
```
⟨*proof*⟩

**end**

The expected height and length of search path defined as non-negative integral. It's easier to prove the recurrence relation of the expected length of the search path using non-negative integrals.

**definition** $NH_N$ **where** $NH_N\ n = \textit{nn-integral}\ (H_N\ n)\ \textit{real}$
**definition** $NR_N$ **where** $NR_N\ n\ u\ l = \textit{nn-integral}\ (R_N\ n\ u\ l)\ \textit{real}$

**lemma** $NH_N$-$EH_N$:
  **assumes** $p \in \{0{<}..{<}1\}$
  **shows** $NH_N\ n = EH_N\ n$
  ⟨*proof*⟩

**lemma** $R_N$-*0* [*simp*]: $R_N\ 0\ u\ l = \textit{return-pmf}\ 0$
  ⟨*proof*⟩

**lemma** $NR_N$-*bounds*:
  **fixes** *u l*::*nat*
  **shows** $NR_N\ n\ u\ l \le l * n + u * NH_N\ n$
⟨*proof*⟩

**lemma** $NR_N$-*recurrence*:
  **assumes** $0 < n\ p \in \{0{<}..{<}1\}$
  **shows** $NR_N\ n\ u\ l = (p * (l + NR_N\ (n - 1)\ u\ l) +$
    $q * (u + (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ l * (\textit{pmf}\ (\textit{binomial-pmf}$
$(n - 1)\ q)\ k))))$
    $/\ (1 - (q\ \widehat{}\ n))$
⟨*proof*⟩

**lemma** $NR_n$-$NH_N$: $NR_N\ n\ u\ 0 = u * NH_N\ n$
⟨*proof*⟩

**lemma** $NR_N$-*recurrence'*:
  **assumes** $0 < n\ p \in \{0{<}..{<}1\}$
  **shows** $NR_N\ n\ u\ l = (p * l + p * NR_N\ (n - 1)\ u\ l +$
    $q * u + q * (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ l * (\textit{pmf}\ (\textit{binomial-pmf}$
$(n - 1)\ q)\ k)))$
    $/\ (1 - (q\ \widehat{}\ n))$
  ⟨*proof*⟩

**lemma** $NR_N$-*l-0*:
  **assumes** $0 < n\ p \in \{0{<}..{<}1\}$
  **shows** $NR_N\ n\ u\ 0 = (p * NR_N\ (n - 1)\ u\ 0 +$
    $q * (u + (\sum k{<}n - 1.\ NR_N\ (k + 1)\ u\ 0 * (\textit{pmf}\ (\textit{binomial-pmf}$
$(n - 1)\ q)\ k))))$
    $/\ (1 - (q\ \widehat{}\ n))$
  ⟨*proof*⟩

**lemma** $NR_N$*-u-0*:
**assumes** $0 < n$ $p \in \{0<..<1\}$
**shows** $NR_N\ n\ 0\ l = (p * (l + NR_N\ (n - 1)\ 0\ l) +$
$q * (\sum k<n - 1.\ NR_N\ (k + 1)\ 0\ l * (pmf\ (binomial\text{-}pmf\ (n - 1)\ q)\ k)))$
$/ (1 - (q\ \hat{}\ n))$
⟨*proof*⟩

**lemma** $NR_N$*-0*[*simp*]: $NR_N\ 0\ u\ l = 0$
⟨*proof*⟩

**lemma** $NR_N$*-1*:
**assumes** $p \in \{0<..<1\}$
**shows** $NR_N\ 1\ u\ l = (u * q + l * p) / p$
⟨*proof*⟩

**lemma** $NR_N$*-*$NR_N$*-l-0*:
**assumes** *n*: $0 < n$ **and** *p*: $p \in \{0<..<1\}$ **and** $u \geq 1$
**shows** $NR_N\ n\ u\ 0 = (u * q / (u * q + l * p)) * NR_N\ n\ u\ l$
⟨*proof*⟩

Assigning 1 as the cost for going up and/or left, we can now show the relation between the expected length of the reverse search path and the expected height.

**definition** $EL_N$ **where** $EL_N\ n = measure\text{-}pmf.expectation\ (R_N\ n\ 1\ 1)\ real$

**theorem** $EH_N$*-*$EL_{sp}$:
**assumes** $p \in \{0<..<1\}$
**shows** $1 / q * EH_N\ n = EL_N\ n$
⟨*proof*⟩

**end**

**thm** *random-skip-list.*$EH_N$*-*$EL_{sp}$[*unfolded random-skip-list.q-def*]
*random-skip-list.*$EH_N$*-bounds′*[*unfolded random-skip-list.q-def*]

**end**

# References

[1] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[2] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449. Springer, 1989.