15-210
Parallel and Sequential Algorithms and Data Structures

Lecture 8

Sets and Tables–II

Synopsis

- How search engines work
- Single-threaded sequences

Building a Search Engine

How do search engines work?

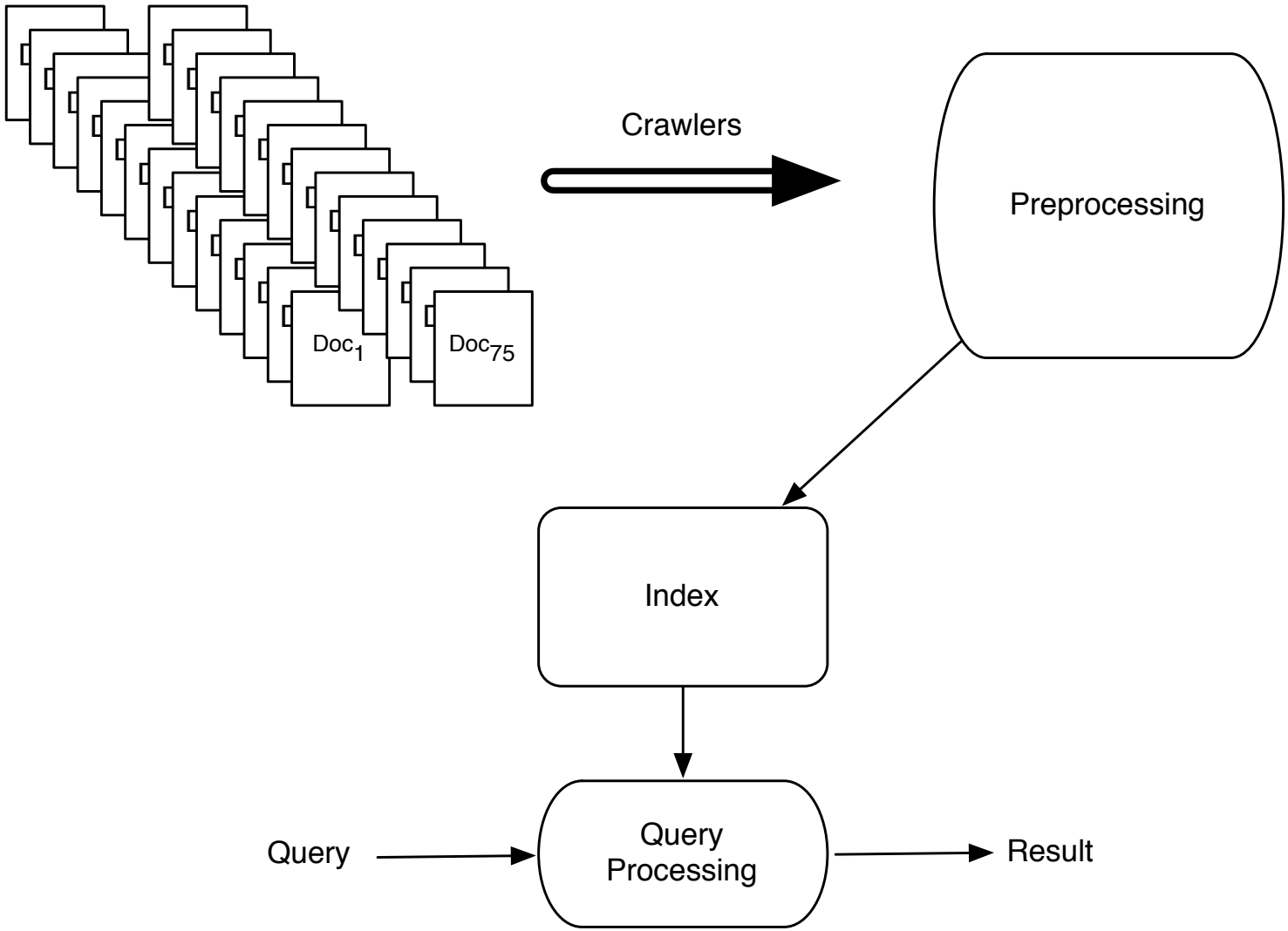
- What are the inputs?
  - (Billions and billions of) documents consisting of "words".
- How do we interact with the search engine
  - (Boolean) Keyword queries
  - List of matching documents (URLS)

How does the search really work?

- User inputs a query (say a couple of words)
- SE starts searching for the words in each document one-by-one
- Returns documents when they match.

- Not really!
  - Not scalable (even for one user)
- Preprocessing

Preprocessing



Plan

- What kinds of queries we want to have.
- What is the interface we want to have.
- How do we implement all these

Queries

- Bingle (:-) supports logical queries on words involving
  - `And`: "15210" `And` "course" `And` "slides"
  - `Or`: "15210" `Or` "15150"
  - `AndNot`: "15210" `AndNot` "Pittsburgh"
- "CMU" `And` "fun" `And` ("courses `Or` "clubs")
- Result would be a list of webpages/documents that match.

The Interface

```
signature INDEX = sig
  type word = string
  type docId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (docId * string) seq -> index

  val find : index -> word -> docList
  val And : docList * docList -> docList
  val AndNot : docList * docList -> docList
  val Or : docList * docList -> docList
  val size : docList -> int
  val toSeq : docList -> docId seq
```

Documents

- Indexing a tweet database.

$T = \langle$ (“jack”, “chess club was fun”),
(“mary”, “I had a fun time in 210 class today”),
(“nick”, “food at the cafeteria sucks”),
(“sue”, “In 217 class today I had fun reading my email”),
(“peter”, “I had fun at nick’s party”),
(“john”, “tiddlywinks club was no fun, but more fun than 218”)
$\rangle$

- “jack” is a document id
- “chess club was fun” is a document

Using the Interface

$T = \langle$ (“jack”, “chess club was fun”),
(“mary”, “I had a fun time in 210 class today”),
(“nick”, “food at the cafeteria sucks”),
(“sue”, “In 217 class today I had fun reading my email”),
(“peter”, “I had fun at nick’s party”),
(“john”, “tiddlywinks club was no fun, but more fun than 218”),
$\rangle$

$f = (\texttt{find}\ (\texttt{makeIndex}(T))) : \texttt{word} \rightarrow \texttt{doclist}$

```
toSeq(And(f "fun", Or(f "class", f "club")))
⇒ ⟨"jack", "mary", "sue", "john"⟩
```

Using the Interface

$T = \langle$ (“jack”, “chess club was fun”),
(“mary”, “I had a fun time in 210 class today”),
(“nick”, “food at the cafeteria sucks”),
(“sue”, “In 217 class today I had fun reading my email”),
(“peter”, “I had fun at nick’s party”),
(“john”, “tiddlywinks club was no fun, but more fun than 218”),
$\rangle$

```
size(AndNot(f "fun", f "tiddlywinks"))
```

$\Rightarrow$ 4

THE MAKEINDEX FUNCTION

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4      Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
5      Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

- What does `tagWords` do?

tagWords(*"jack"*, *"chess club was fun"*)

⇒ ⟨(*"chess"*,*"jack"*),(*"club"*, *"jack"*), (*"was"*, *"jack"*), (*"fun"*, *"jack"*)⟩

THE PAIRS FUNCTION

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4      Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
5      Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

- What does `Pairs` do?

$$\texttt{Pairs} = \langle(\text{“chess”},\text{“jack”}),(\text{“club”},\text{“jack”}),(\text{“was”},\text{“jack”}),\ (\text{“fun”},\text{“jack”}),(\text{“I”},\text{“mary”}),(\text{“had”},\text{“mary”}),\ (\text{“fun”},\text{“mary”}),\ldots\rangle$$

The collect function

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4      Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
5      Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

- What does `collect` do?

$$
\begin{aligned}
\texttt{Words} = \{&(\text{“}a\text{”} \mapsto \langle \text{“}mary\text{”} \rangle),\\
&(\text{“}at\text{”} \mapsto \langle \text{“}mary\text{”}, \text{“}peter\text{”} \rangle),\\
&\ldots\\
&(\text{“}fun\text{”} \mapsto \langle \text{“}jack\text{”}, \text{“}mary\text{”}, \text{“}sue\text{”}, \text{“}peter\text{”}, \text{“}john\text{”} \rangle),\\
&\ldots
\end{aligned}
$$

Final Touches

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4      Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
5      Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

- What is happening here?
- Sequences are converted to tables.

makeIndex Costs

```
1  fun makeIndex(docs) =
2  let
3    fun tagWords(id, str) = ⟨(w, id) : w ∈ tokens(str)⟩
4      Pairs = flatten⟨tagWords(d) : d ∈ docs⟩
5      Words = Table.collect(Pairs)
6  in
7    {w ↦ Set.fromSeq(d) : (w ↦ d) ∈ Words}
8  end
```

- Assuming tokens have a upper bound on length
  - $W_{makeIndex}(n) \in O(n \log n)$, $S_{makeIndex} \in O(\log^2 n)$
  - What does n represent?

Rest of the Interface

fun $\texttt{find}\ T\ v = \texttt{Table.find}\ T\ v$

fun $\texttt{And}(s_1, s_2) = s_1 \cap s_2$

fun $\texttt{Or}(s_1, s_2) = s_1 \cup s_2$

fun $\texttt{AndNot}(s_1, s_2) = s_1 \setminus s_2$

fun $\texttt{size}(s) = |s|$

fun $\texttt{toSeq}(s) = \texttt{Set.toSeq}(s)$

Single-threaded Array Sequences

- Updating an array sequence in an imperative language takes $O(1)$ work.
- In a functional setting, everything is persistent.
- An update to a sequence of n elements needs
  - $O(n)$ work for `arraySequence` implementation to copy and update.
  - $O(\log n)$ work for `treeSequence` implementation (because of substructure sharing)
- Interfaces do not provide functions for updating a single position.
  - to discourage sequential (and expensive) computation.

Single-threaded Array Sequences

- A *map* can be implemented as follows

  fun *map* f $S =$
  $\quad$ *iter* (**fn** $((i, S'), v) \Rightarrow (i+1,$ *update* $(i, f(v))$ $S'))$
  $\quad\quad\quad\quad$ $(0, S)$
  $\quad\quad\quad\quad$ S

- Iterates n times ($i = 0, \ldots n-1$)
- and updates the value S_i with $f(S_i)$.
- `arraySequence`: Each update will do $O(n)$ work for a total $O(n^2)$ work
- `treeSequence`: Each update will do $O(\log n)$ work for a total $O(n \log n)$ work.

Single-Threaded Sequences

- A new ADT: Single Threaded Sequence: `stseq`
- Useful when a bunch of items need to be updated.
- Straigthforward interface
- Cost specification imply non-functional stuff under the hood!

STSEQ INTERFACE AND COSTS

	Work	Span
`fromSeq(S)` : α `seq` $\rightarrow$ α `stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST)` : α `stseq` $\rightarrow$ α `seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i : α `stseq` $\rightarrow$ `int` $\rightarrow$ α Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i, v) S : (`int` $\times$ α) $\rightarrow$ α `stseq` $\rightarrow$ α `stseq` Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
`inject` I S: (`int` $\times$ α) `seq` $\rightarrow$ α `stseq` $\rightarrow$ α `stseq` For each $(i, v) \in I$ replaces the i^{th} element of S with v.	$O(\|I\|)$	$O(1)$

- Cost bounds for `nth` and `update` only valid for the "current" version of the sequence.

Map with stseq

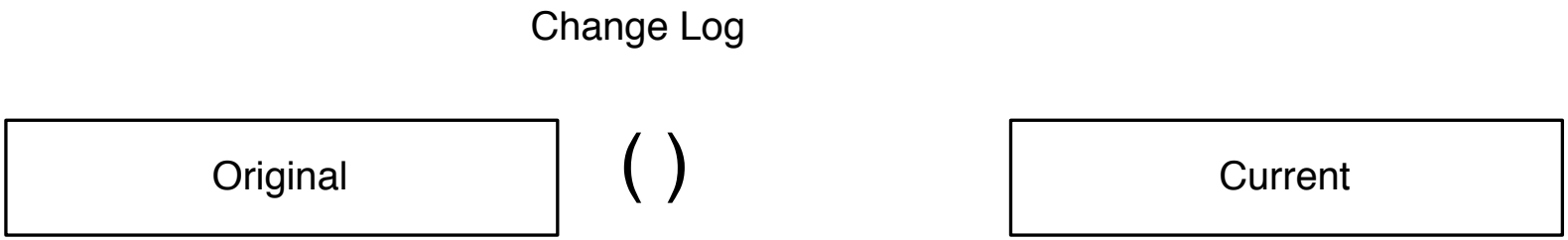
```
1  fun map f S = let
2      S' = StSeq.fromSeq(S)
3      R = iter
4            (fn ((i, S''), v) ⇒ (i + 1, StSeq.update (i, f(v)) S''))
5            (0, S')
6            S'
7  in
8     StSeq.toSeq(R)
9  end
```

- Overall work and span is $O(n)$ (Why?)
- Multiple updates can be done in $O(n)$ time.

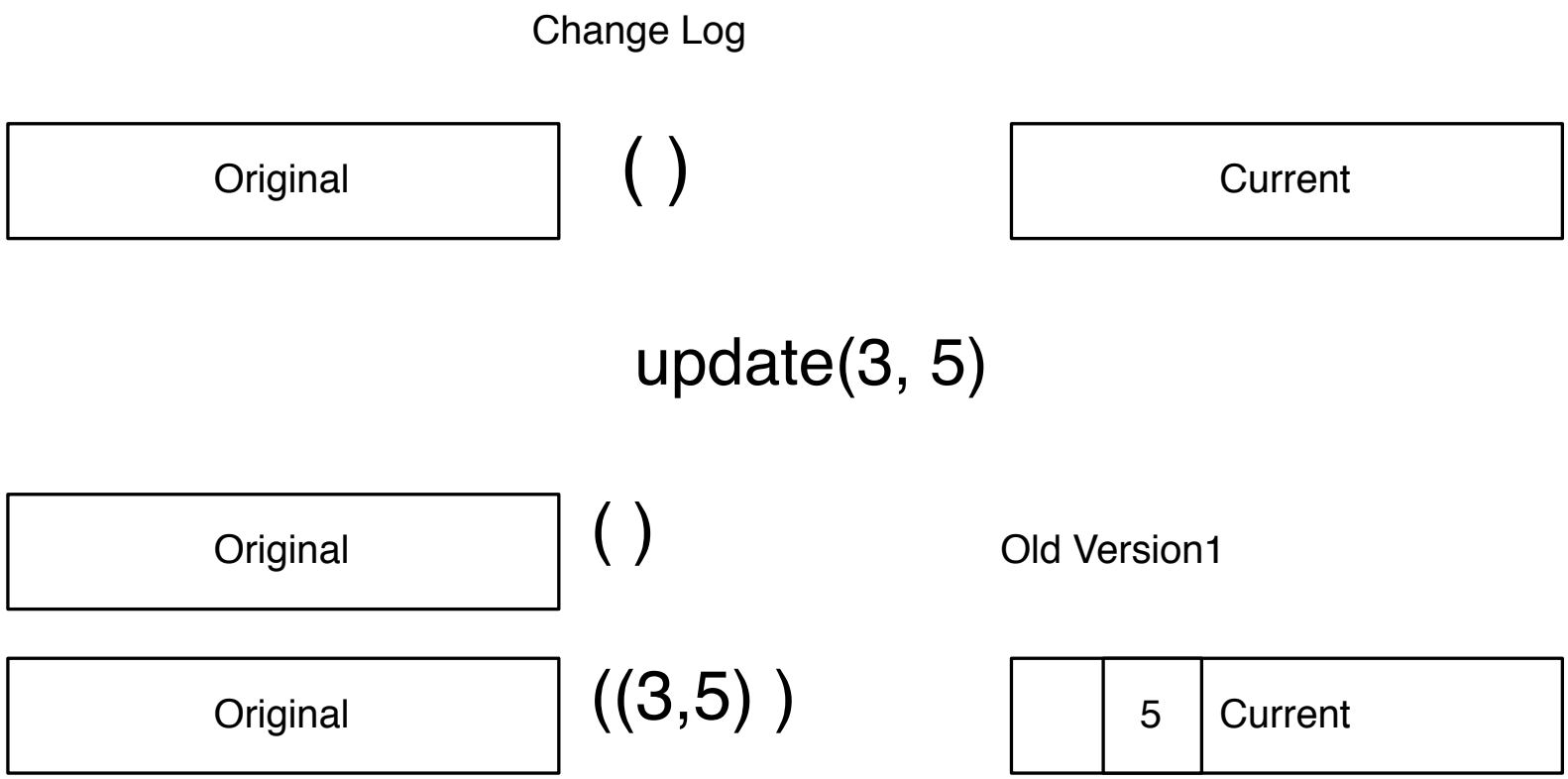
Implementing stseq

- Keep two full copies of the sequence
  - Original and Current
  - We keep a change log: updates to the original to get Current.
- When Current is updated
  - We create a "new" Current with the update, and update change log.
  - Mark the previous version as old, remove its Current and but keep the old change log.
- Any item from the current version is accessible in constant work.
- Any item from the any previous version is accessible but needs more work.

Implementing stseq

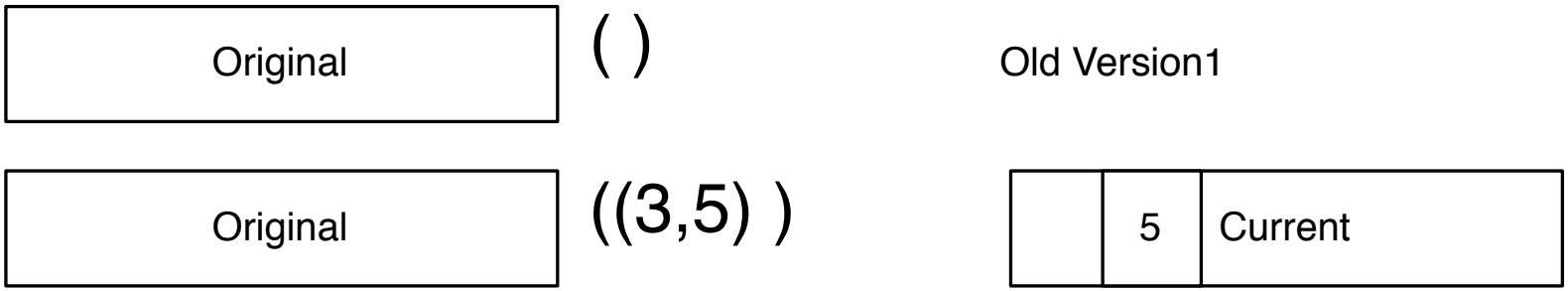


Implementing stseq

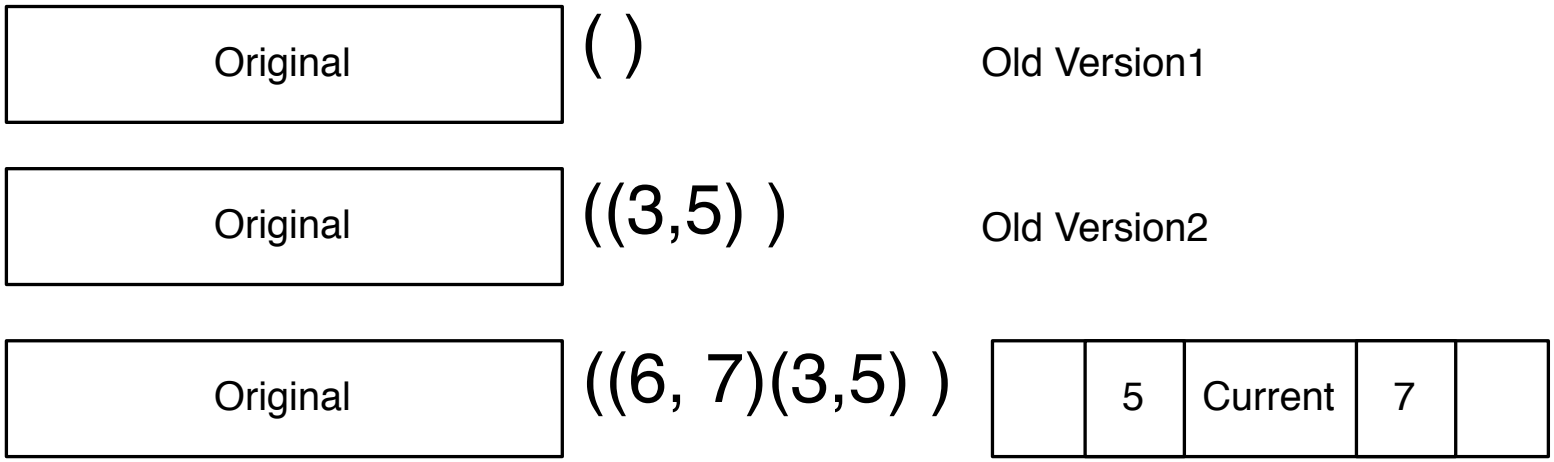


- There really is only one copy of the Original.
- All point to that copy.

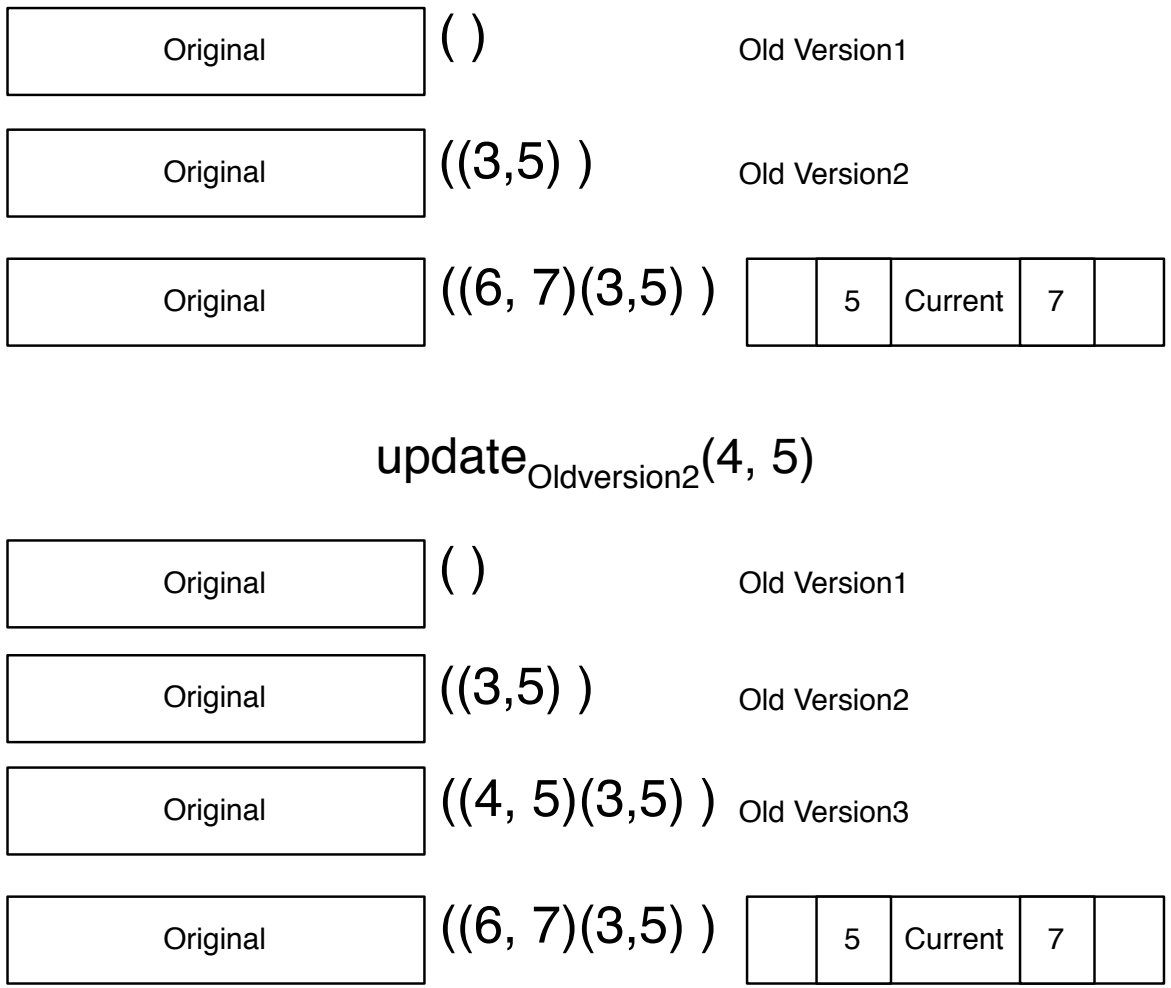
Implementing stseq



update(6, 7)



Implementing stseq



Summary

- How search engines work
- Single-threaded sequences