Lecture 8

Text Indexing

Rasmus Pagh

in part based on slides by
S. Srinivasa Rao and Paolo Ferragina

Today's lecture

- B^+-trees and strings.
- Inverted indexes.
- Fingerprinting - Rabin-Karp.
- Full-text indexing
  - Suffix arrays and suffix trees.
  - String B-trees.

Why is string data interesting ?

Ubiquitous:

- Digital libraries (studied a lot already in the '80s).
- Product catalogues, e-commerce websites (this is what got the DB world interested in text).
- Electronic white and yellow pages (Apptus guest lecture)
- Specialized information (e.g. Genomic or Patent dbs)
- Web page repositories (search engines)
- ...

String collections grow at a staggering rate:

- ...10s of Tb textual data on the web
- ...10s of Gb of base pairs in the genomic DBs

The need for indexing

- Need special facilities, not just equality comparison.
- Brute-force scanning is usually not a viable approach.
- Indexing allows fast "simple" searches
- Can use multiple simple searches for complex queries

An information retrieval (IR) system also encompasses many aspects we will not go into:

- IR models
- Ranking algorithms
- Query languages and operations
- ...

Strings in a B-tree

- Allows searching for strings with a given prefix.
- Does not help finding, say, a string containing a given word.
- Even when a B-tree is appropriate, string keys can be problematic:
  - Reduce fan-out, increase depth of tree
  - Prefix compression (i.e., omit part of key shared with the previous key) may help, but not always.
- Motivates looking at special indexes.

Two families of indexes

Types of data

Linguistic or tokenizable text
Raw sequence of characters or bytes

DNA sequences
Audio-video files
Executables

Types of query

Word-based query
Character-based query

Exact word
Word prefix or suffix
Phrase

Arbitrary substring
Complex matches

Two indexing approaches :

- Word-based indexes, here a concept of "word" must be devised !
  - » Inverted files, Signature files or Bitmaps.
- Full-text indexes, no constraint on text and queries !
  - » Suffix Array, Suffix tree, Hybrid indexes, or String B-tree.

Word-based: Inverted files

(or lists, or indexes)

Doc #1

Now is the time for all good men to come to the aid of their country

Doc #2

It was a dark and stormy night in the country manor. The time was past midnight

Vocabulary

Term	N docs	Tot Freq
a	1	1
aid	1	1
all	1	1
and	1	1
come	1	1
country	2	2
dark	1	1
for	1	1
good	1	1
in	1	1
is	1	1
it	1	1
manor	1	1
men	1	1
midnight	1	1
night	1	1
now	1	1
of	1	1
past	1	1
stormy	1	1
the	2	4
their	1	1
time	2	2
to	1	2
was	1	2

Occurrences

Doc #	Freq
2	1
1	1
1	1
2	1
1	1
1	1
2	1
2	1
1	1
1	1
2	1
1	1
2	1
2	1
1	1
2	1
2	1
1	1
1	1
2	1
2	1
1	2
2	2
1	1
1	1
2	1
1	2
2	2

2

☑ Query answering is a two-phase process: midnight AND time

Observations on inverted files

- Can be implemented directly in the relational model.
- Use standard index on vocabulary and occurrences.
- Very efficient for single-word search.
- When searching for multiple words, an intersection (i.e. join) is needed.
  - Time depends on the number of occurrences of each word.
- Size will often be smaller than the indexed strings. (Why?)

Refinements to inverted files

- Keep track of locations of words - e.g. allows higher priority to strings where words occur close together.
- Non-exact matches of words
  - Prefix matches, e.g. Ras*
  - General regular expressions
- Ranking mechanisms for results
  - Often, need only ”top k”.

- Next: Full-text indexes - allow searches for any substring (e.g. of a word).

Fingerprinting strings

[Karp-Rabin, 1987]

- Consider a string s to be a sequence of numbers $a_0, a_1, \ldots, a_n$.
- Corresponds to a polynomial
  $p_s(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$
- Value for a given x can be computed with 2n arithmetic operations. (How?)
- <u>Lemma</u>: Two degree n polynomials have the same value for at most n values of x.
- <u>Idea</u>: Choose x at random from a large set, then $p_s(x)$ is unique for each string with high probability.

Problem session

- As stated, the technique works, but is inefficient!
- Try to figure out why.
- If you can, propose a better way.

Fingerprinting strings

[Karp-Rabin, 1987]

- Central observation: If all computation is done modulo a fixed prime p > x, then everything works.
- Further observations:
  - Can compute fingerprints of all prefixes of a string in linear time.
  - Can compute fingerprints of all subtrings of length k in linear time (not k times linear). (BigTable uses this in the compression algorithm.)
  - With fingerprints, we get fast *equality* search: Use them as B-tree keys.

Full-text indexes

Their need is pervasive:

- Raw data: DNA sequences, Audio-Video files, ...
- Linguistic texts: data mining, statistics, ...
- Vocabulary for Inverted Lists
- Xpath queries on XML documents
- Intrusion detection, Anti-viruses, ...

Classes of indexes:

- Suffix array, Suffix tree (variants)
- Multi-level indexes: Short Pat array
- B-tree based data structures: Prefix B-tree, String B-tree

Terminology

- An alphabet, denoted Σ is a set of (ordered) characters.
- A string S is an array of characters, S[1,n] = S[1] S[2] … S[n].
- S[i,j] = S[i] … S[j] is a substring of S.
- S[1,j] is a prefix of S; S[i,n] is a suffix of S.
- Σ^* denotes all strings over alphabet Σ.
- Lexicographic order:
  - Example: For Σ = { a, b, c, …, z }, where a < b < c < … < z, the lexicographic order is the same as in a dictionary.

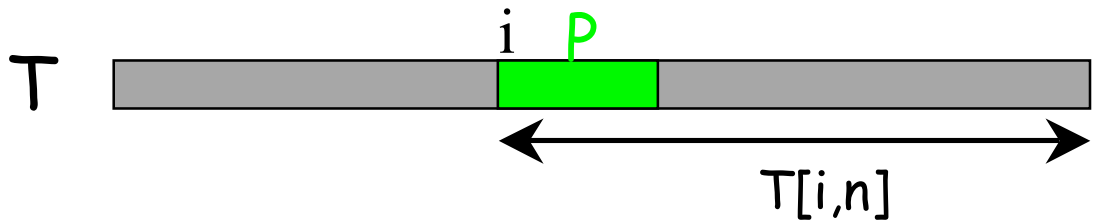
SUF(T) = Sorted set of suffixes of T

Indexed string matching problem

- T is a set of K strings in Σ^*
  - N is the total length of all strings in T.
- String matching query on T:
  - Given a pattern P find all occurrences of P in T.
- *Full-text index* : Store T in a data structure that supports string matching queries.
- *Dynamic full-text index*: Supports also insertions and deletions of strings in the full-text index.

A simple but crucial observation

Pattern P[1,p] occurs at position i of T[1,n]
iff P[1,p] is a prefix of the suffix T[i,n]



Occurrences of P in T = All suffixes of T having P as a prefix

T = This is a visual example
This is a visual example
This is a visual example

3,6,12

Can transform the string matching problem to a "prefix matching problem" over all the suffixes.

Suffix Array [Manber-Myers, 1990]

Suffix array: an array of pointers to all the suffixes in the text in their lexicographic order.

$\Theta(N^2)$ space

5

T = mississippi#

suffix pointer

SA	SUF(T)
12	#
11	i#
8	ippi#
5	issippi#
2	ississippi#
1	mississippi#
10	pi#
9	ppi#
7	sippi#
4	sissippi#
6	ssippi#
3	ssissippi#

Suffix Array

- SA: array of ints, 4N bytes
- Text T: N bytes

⇨ 5N bytes of space occupancy

Two key properties [Manber-Myers, 1990]

Prop 1. All suffixes in SUF(T) having prefix P are contiguous.
Prop 2. Starting position is the lexicographic position of P.

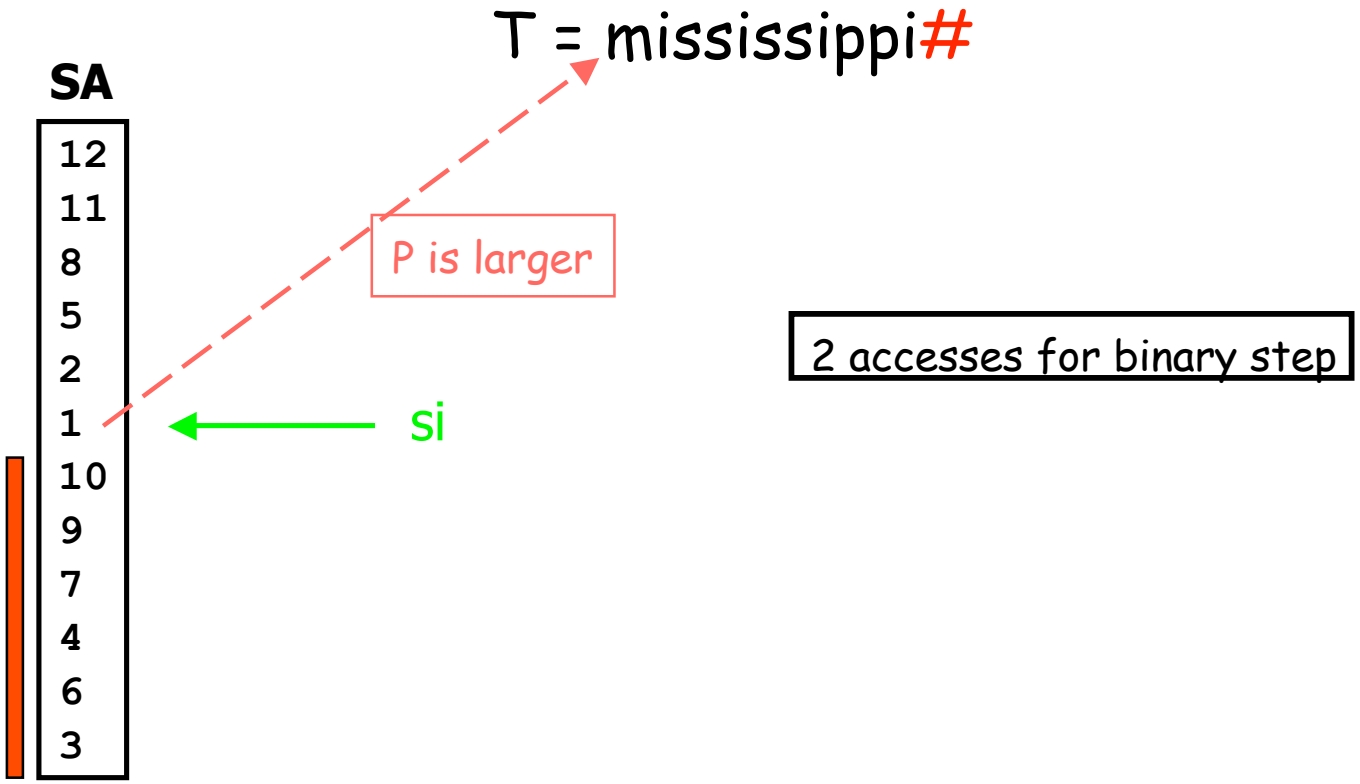
T = mississippi#

SA	SUF(T)
12	#
11	i#
8	ippi#
5	issippi#
2	ississippi#
1	mississippi#
10	pi#
9	ppi#
7	sippi#
4	sissippi#
6	ssippi#
3	ssissippi#

P=si

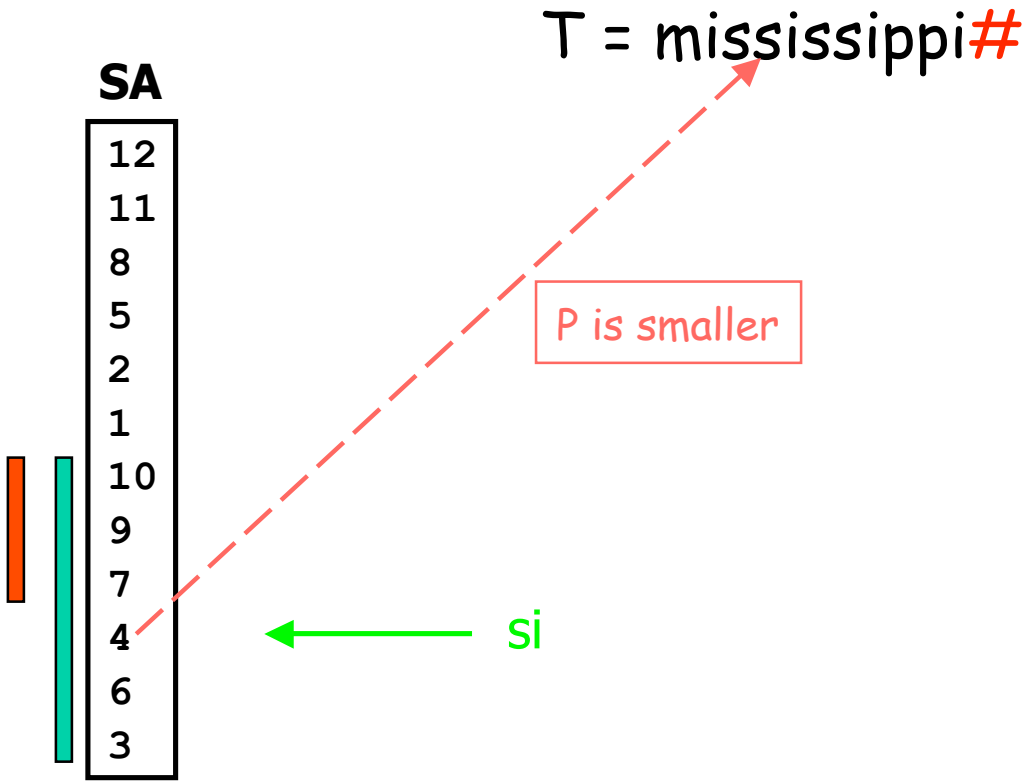
Searching in a Suffix Array [Manber-Myers, 1990]

Indirected binary search on SA: $O(p \log_2 N)$ time



Searching in a Suffix Array [Manber-Myers, 1990]

Indirected binary search on SA: $O(p \log_2 N)$ time



Listing the occurrences [Manber-Myers, 1990]

Brute-force comparison: O(p x occ) time

T = mississippi#

4 6 7

SA

12
11
8
5
2
1
10
9
7
4
6
3

occ=2

si

sippi — P is a prefix

sissippi — P is a prefix

issippi — P is not a prefix

Suffix Array search

- $O(p(\log_2 N + occ))$ time
- $O(\log_2 N + occ)$ in practice

External memory

- Simple disk paging for SA
- $O((p/B)(\log_2 N + occ))$ I/Os

$+$ $\log_B N$ occ/B

Problem session

- Draw the suffix array for the string PAPAYAS.
- How will a search for the string PAY proceed?

Output-sensitive retrieval

Lcp[1,n-1] stores the longest-common-prefix between suffixes adjacent in SA

T = mississippi#

4 6 7

Lcp	SA	SUF(T)
	12	#
0	11	i#
0	8	ippi#
1	5	issippi#
4	2	ississippi#
0	1	mississippi#
0	10	pi#
1	9	ppi#
0	7	sippi#
2	4	ssippi#
1	6	ssippi#
3	3	ssissippi#

occ=2

P=si

Compare against P

Scan Lcp until Lcp[i] < p

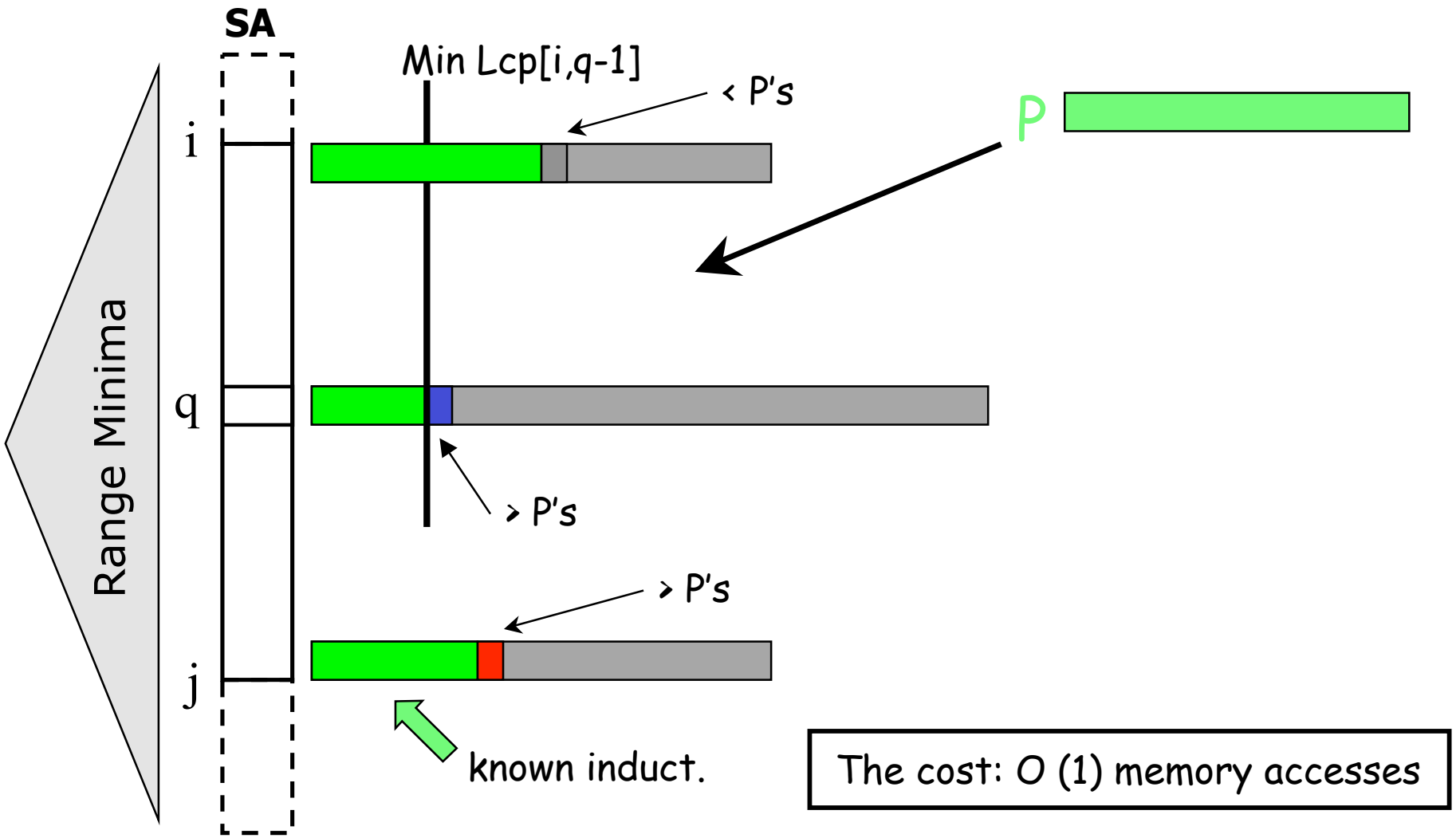
Suffix Array search

- O ((p/B) $\log_2$ N + (occ/B)) I/Os
- 9 N bytes of space

+ : incremental search

Incremental search (case 1)

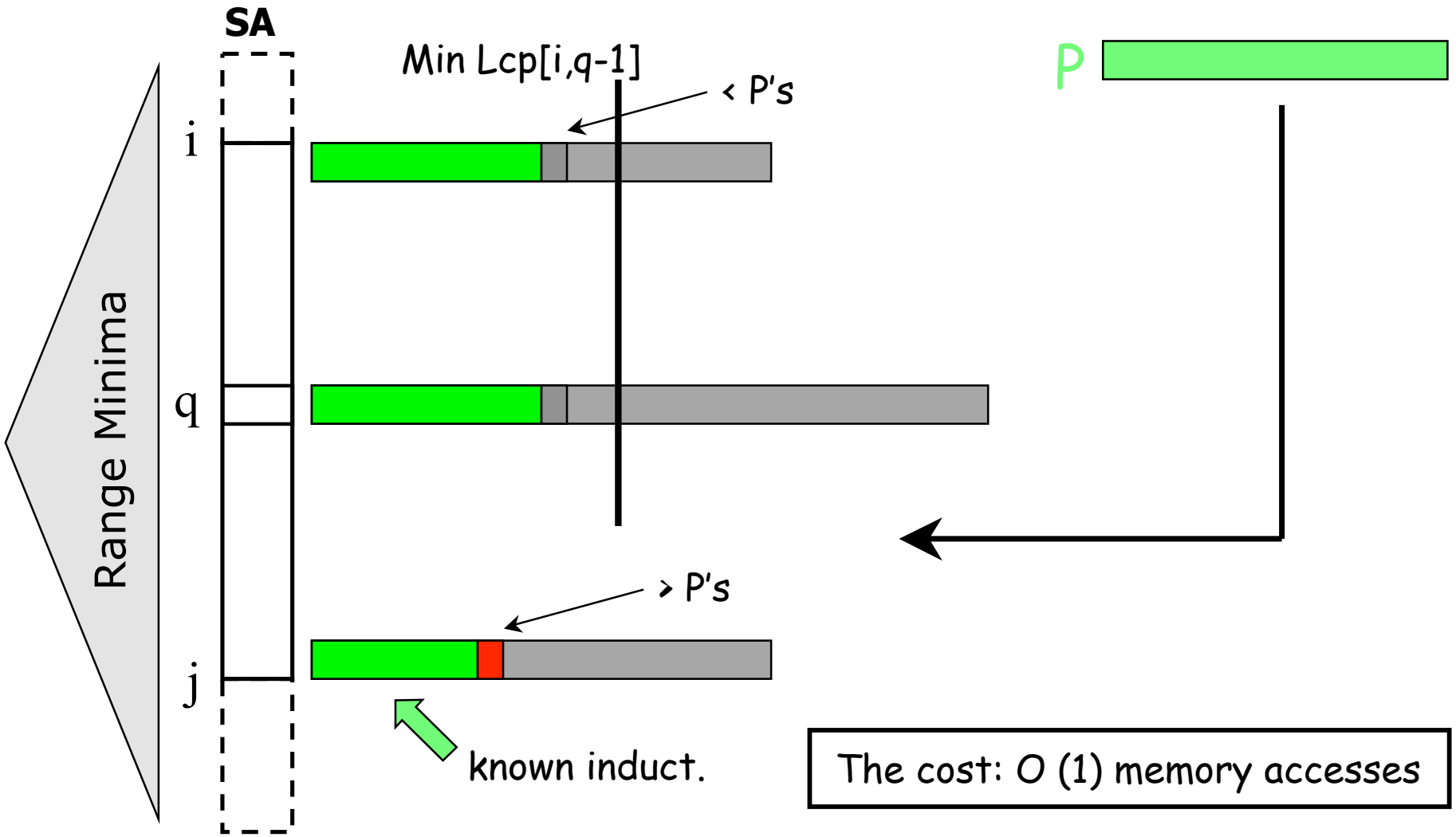
Incremental search using the LCP array: no rescanning of pattern chars



The cost: O (1) memory accesses

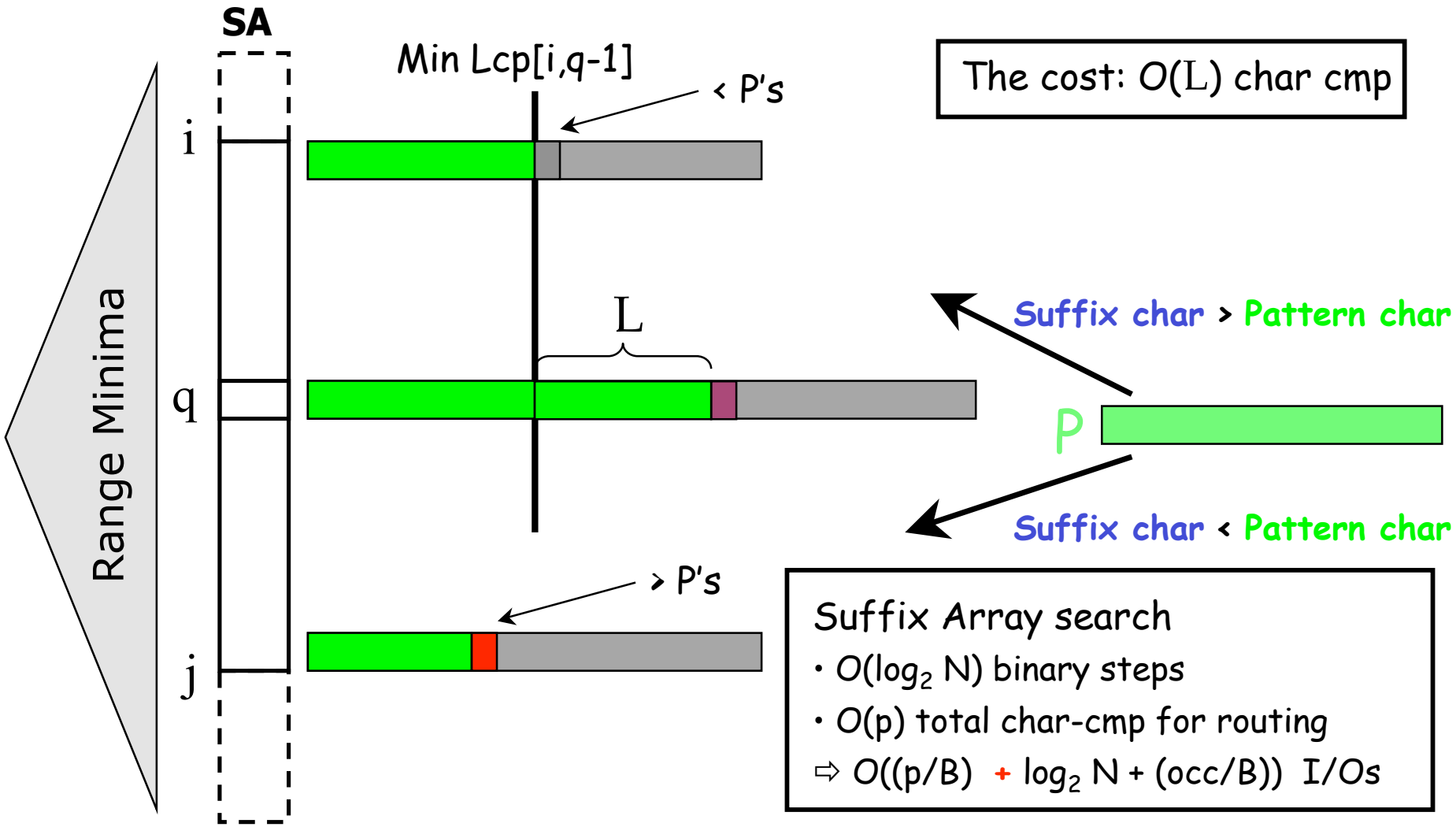
Incremental search (case 2)

Incremental search using the LCP array: no rescanning of pattern chars



Incremental search (case 3)

Incremental search using the LCP array: no rescanning of pattern chars



Suffix Array search

- $O(\log_2 N)$ binary steps
- $O(p)$ total char-cmp for routing

⇨ $O((p/B) + \log_2 N + (occ/B))$ I/Os

Suffix arrays and updates

- Suppose there is a (small) change in the text. The suffix array must be updated.
- Problem: Most of the suffixes are likely to have changed.
- Possible solution:
  - Bound then length of comparison by k.
  - Suffixes that are equal in the first k characters may occur in any order.
  - Use a B-tree to store suffix order.
- **Problem session**:
  - Argue that after an update we need to move at most k suffixes in the tree.
  - Give an example where this leads to more expensive queries for string length > k.

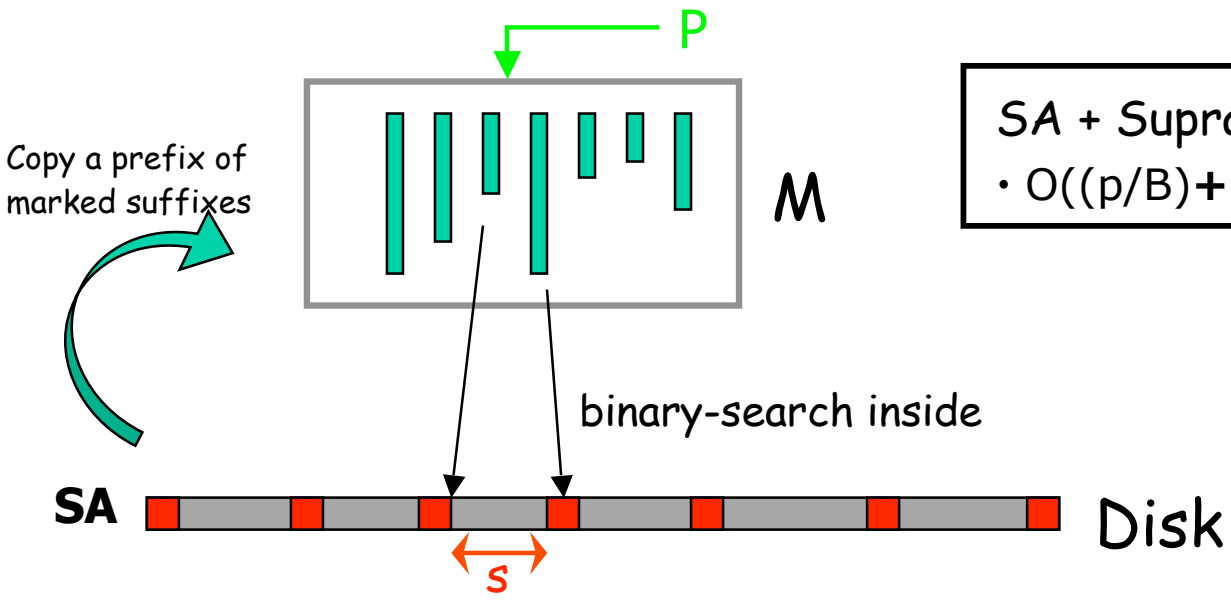
Summary: suffix array

[Manber-Myers, 1990]

- Space: O(N)
- String matching: O(p +log N + occ)
- Can be constructed O(N log N) time.
- Recently: Simple O(N) time algorithms
  - My favorite: Kärkkainen-Sanders, 2003.
  - Also works well on external memory.
- Static.
  - Can be adapted to be dynamic, but then performance guarantees are worse.

Hybrid Index: Short Pat array

Exploit internal memory: sample the suffix array and copy something in memory



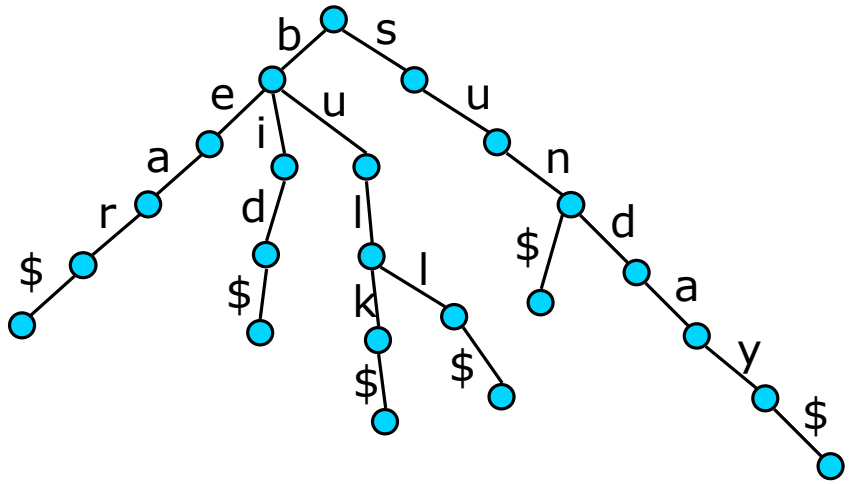
SA + Supra-index

- $O((p/B)+\log_2(N/s)+(occ/B))$ I/Os

☑Parameter s depends on M and influences both performance and space !!
(only a heuristic)

Tries

- *Trie* (name from the word "re*trie*val"): a data structure for storing a set of strings
  - Let's assume that all strings end with "$" (not in Σ)



Set of strings: {**bear, bid, bulk, bull, sun, sunday**}

Tries

- Properties of a ***trie***:
  - A multi-way tree.
  - Each ***node*** has from 1 to Σ+1 children.
  - Each ***edge*** of the tree is labeled with a character.
  - Each ***leaf*** node corresponds to the stored string, which is a concatenation of characters on a path from the root to this node.

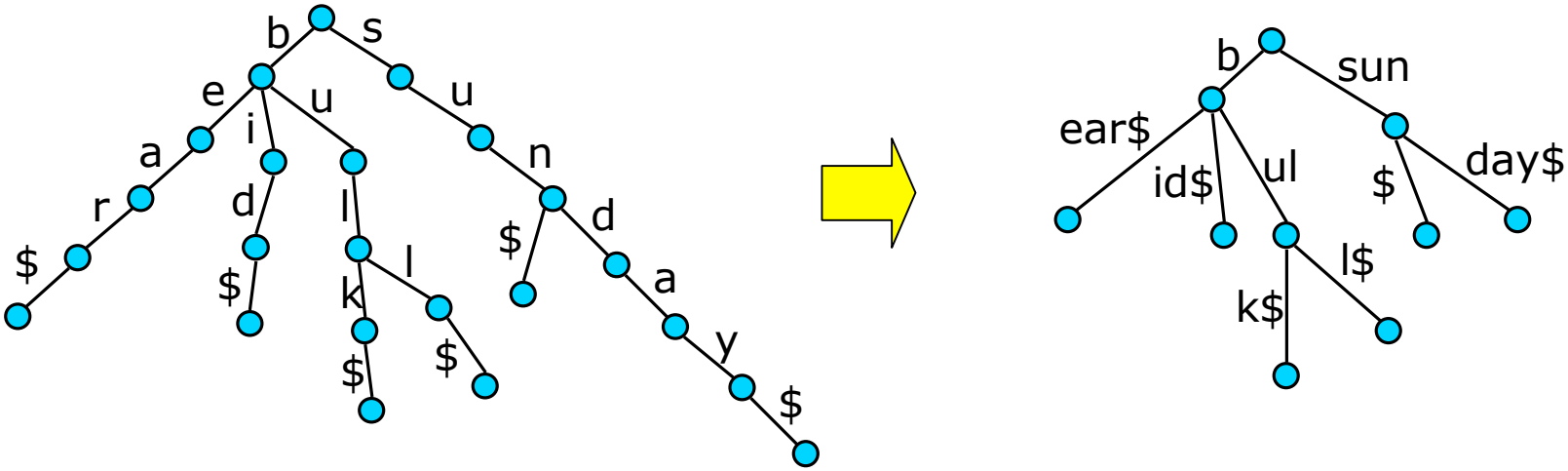
Analysis of the Trie

Given k strings of total length N:

- Size:
  - O(N) in the worst-case
- Search, insertion, and deletion (string of length *m*):
  - O(m) (assuming Σ is constant)
- Observation:
  - Having chains of one-child nodes is wasteful

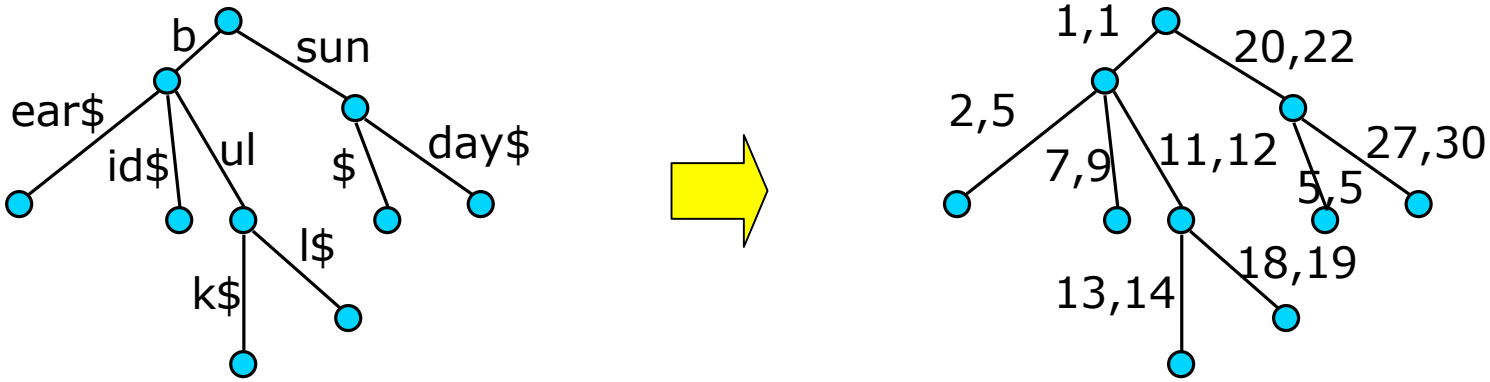
Compact Tries

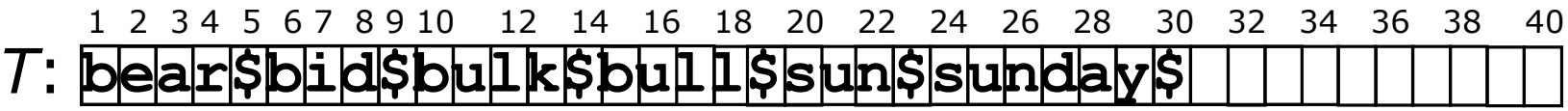
- *Compact Trie*:
  - Replace a *chain* of one-child nodes with an edge labeled with a string
  - Each non-leaf node (except root) has at least two children



Compact Tries

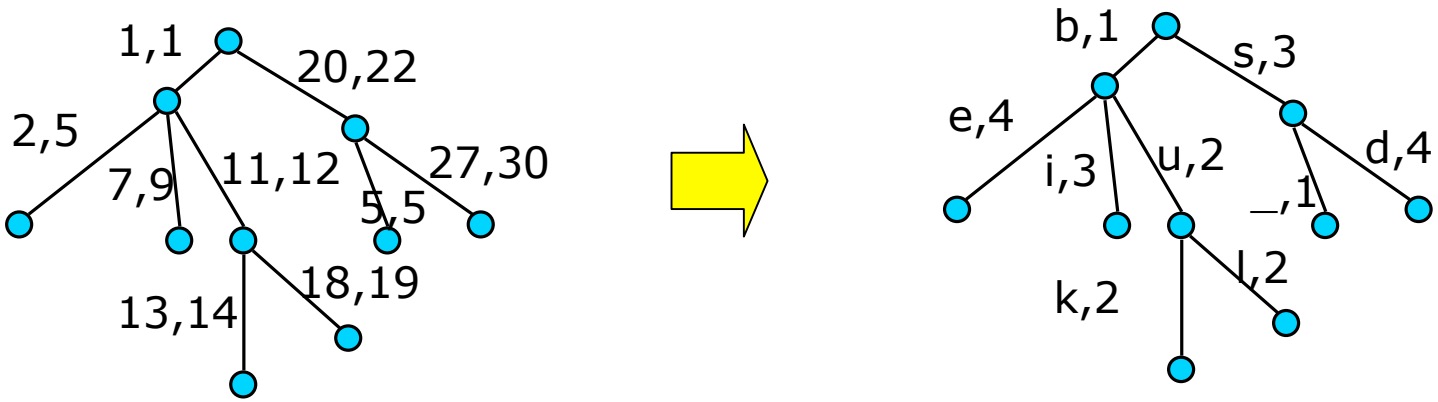
- Implementation:
  - Strings are external to the structure in one array, edges are labeled with indices in the array (*from*, *to*)
  - Improves the space to O(k)

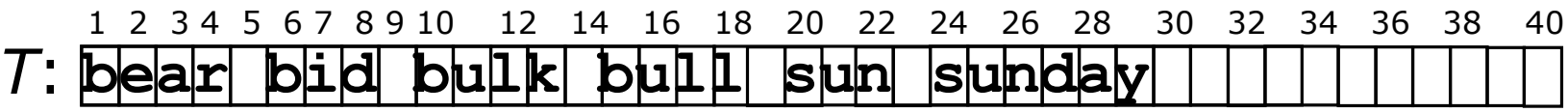




Patricia Tries

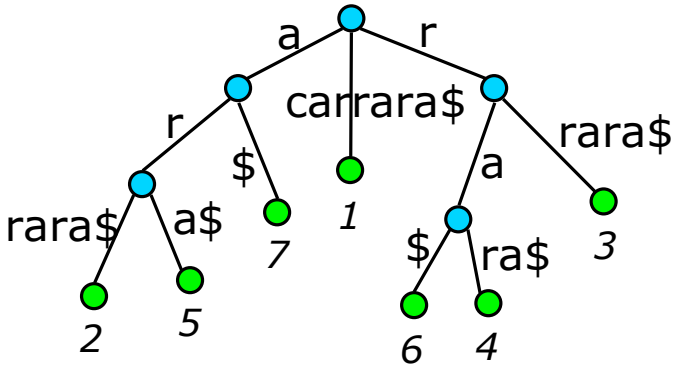
- *Patricia trie*:
  - a compact trie where each edge's label (*from*, *to*) is replaced by (*T*[*from*], *to* – *from* + *1*)

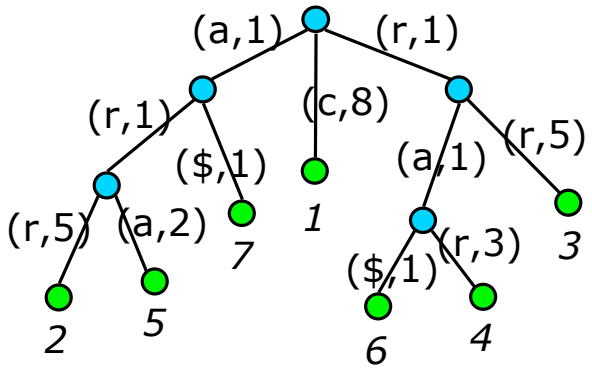




Suffix Trees [McCreight, 1976]

- *Suffix tree* – a compact trie (or similar structure) of all suffixes of the text
  - Patricia trie of suffixes is sometimes called a *Pat tree*





1 2 3 4 5 6 7 8

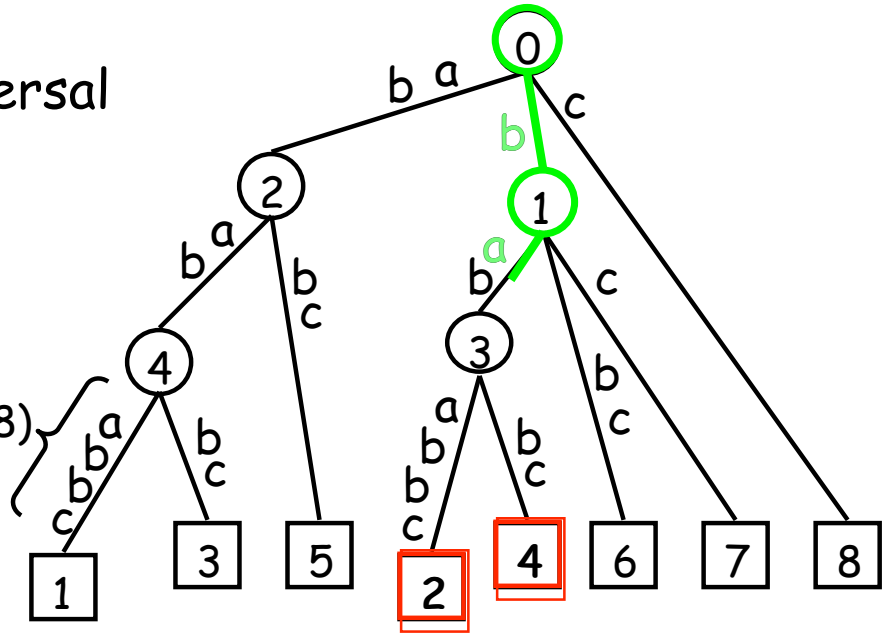
c	a	r	r	a	r	a	$

Search in suffix trees

P = ba → Search is a path traversal

O(p) time and O(occ) time

O(N) space



T = abababbc#

1 3 5 7 9

What about ST in external memory ?

- Unbalanced tree topology
- Updates

} Packing ?!

- Large space ~ 15N } CPAT tree ~ 5N on average

Problem session

- When searching in a suffix tree, why is it not a problem that some nodes may have large degree?

- Draw the suffix tree for the string PAPAYAS.
- How will a search for the string PAY proceed?

Summary: compact trie

K strings of total length N:

- Space: O(K).
- String matching queries: O(p + occ) time.
- Can be constructed O(N) time.
- Update(S): O(|S|) time.

Summary: suffix tree

For a string of length n:

- Space: O(n).
- String matching queries: O(p + occ).
- Can be constructed O(n) time.
- Static
  - Again, bounding the length of comparison can be done to allow reasonably fast updates.
- Question: What are the pros and cons of suffix trees, versus suffix arrays?

The String B-tree

[Ferragina-Grossi, 95]

The prologue

We are left with many open issues:

- Suffix Array: updates
- Hybrid: Heuristic tuning of the performance
- Suffix tree: difficult packing and $\Omega(p)$ I/Os

B-tree is ubiquitous in large-scale applications:

- Atomic keys: integers, reals, ...
- Prefix B-tree: bounded length keys ($\leq$ 255 chars)

Suffix trees + B-trees ? → String B-tree [Ferragina-Grossi, 95]

☑ Index unbounded length keys

☑ Good worst-case I/O-bounds in search and update

Some considerations

Strings have arbitrary length:

- Disk page cannot ensure the storage of $\Theta(B)$ strings
- M may be unable to store even one single string

String storage:

- Pointers allow to fit $\Theta(B)$ strings per disk page
- String comparison needs disk access and may be expensive

String pointers organization seen so far:

- Suffix array: simple but static and not optimal
- Patricia trie: sophisticated and very efficient (optimal ?)

Recall the problem: T is a string collection

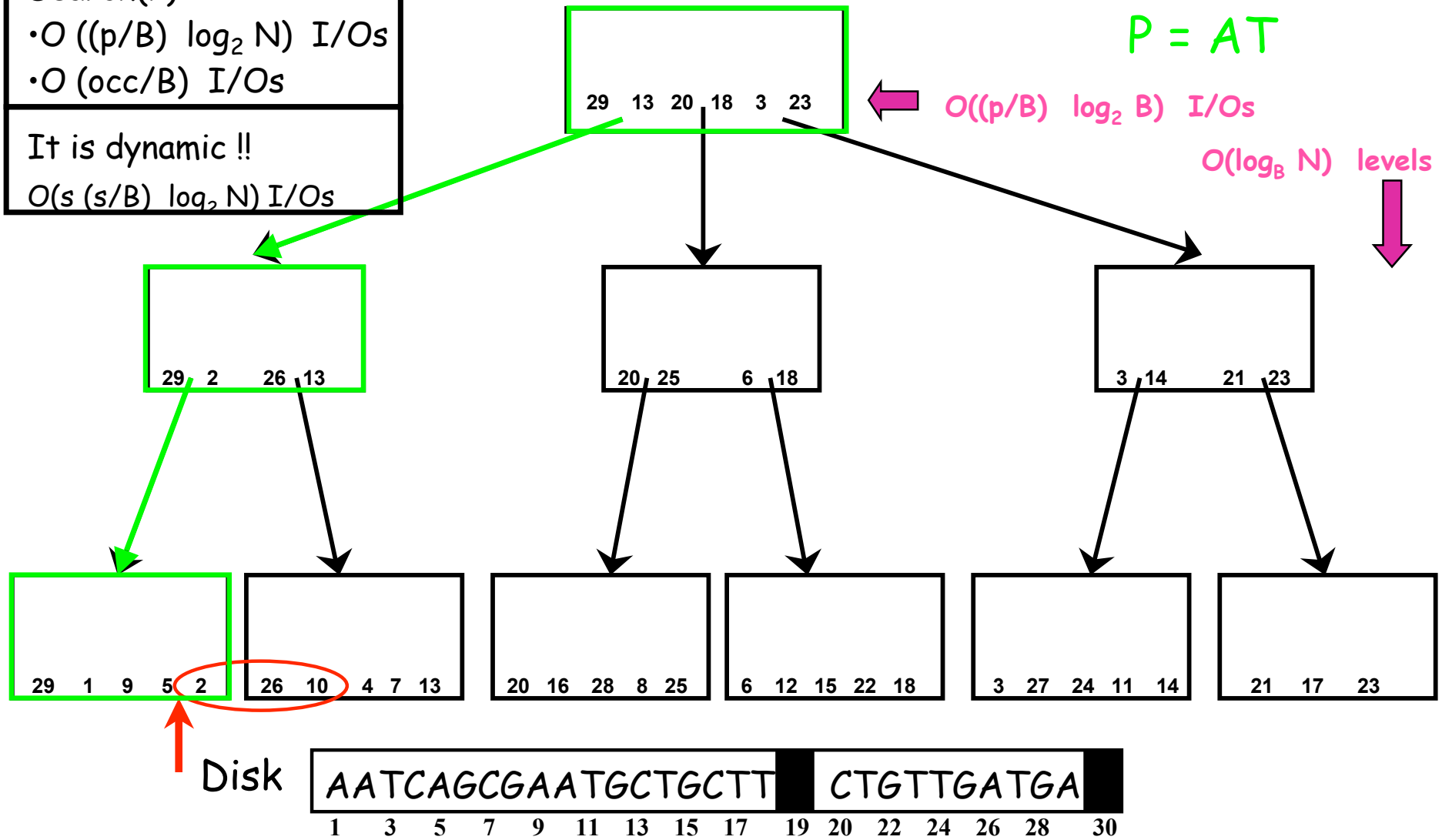
- ✓ Search(P[1,p]): retrieve all occurrences of P in T's strings
- ✓ Update(S[1,s]): insert or delete a text S from T

1st step: B-tree on string pointers

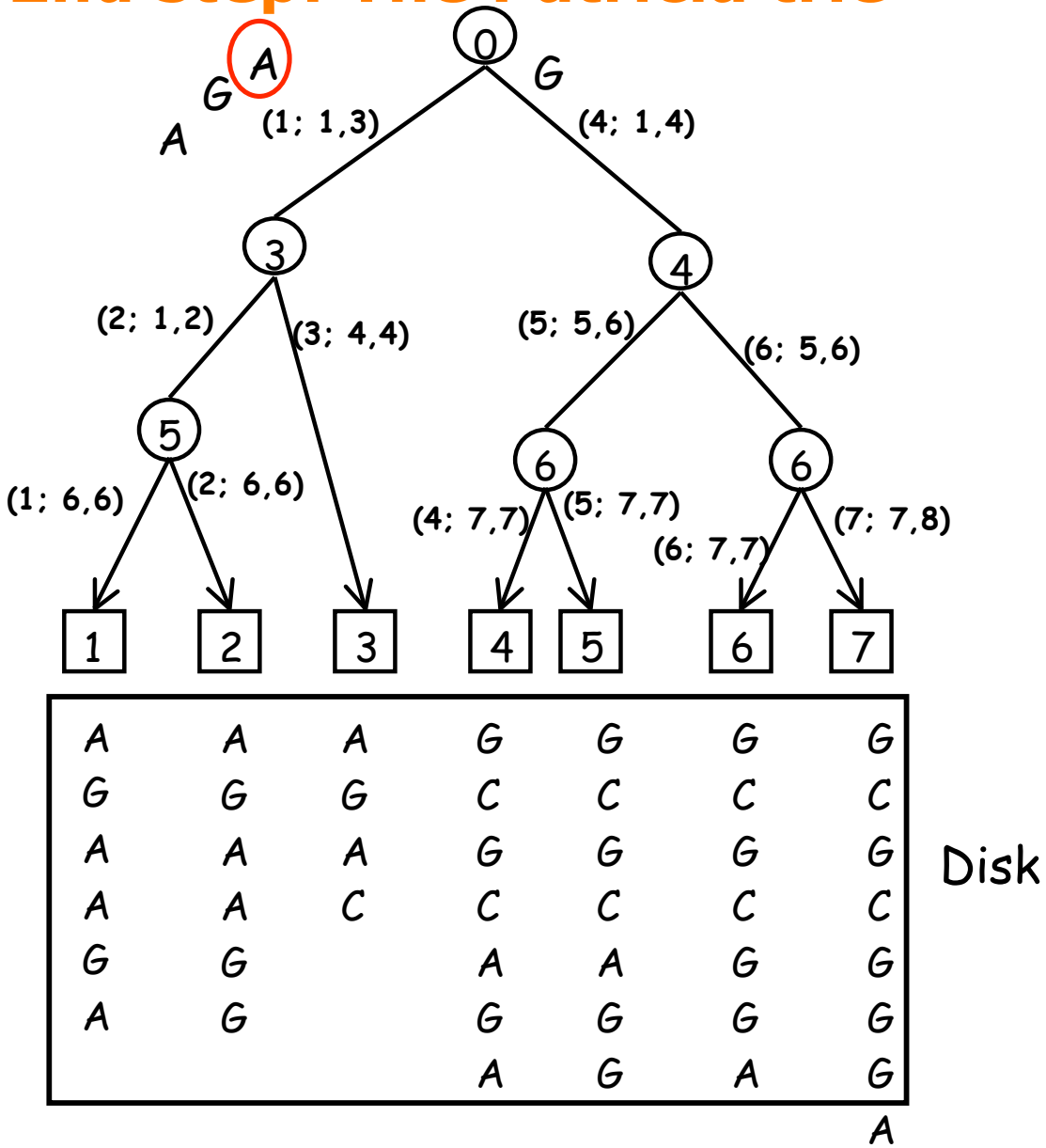
Search(P)
- O ((p/B) $\log_2$ N) I/Os
- O (occ/B) I/Os

It is dynamic !!

O(s (s/B) $\log_2$ N) I/Os



2nd step: The Patricia trie

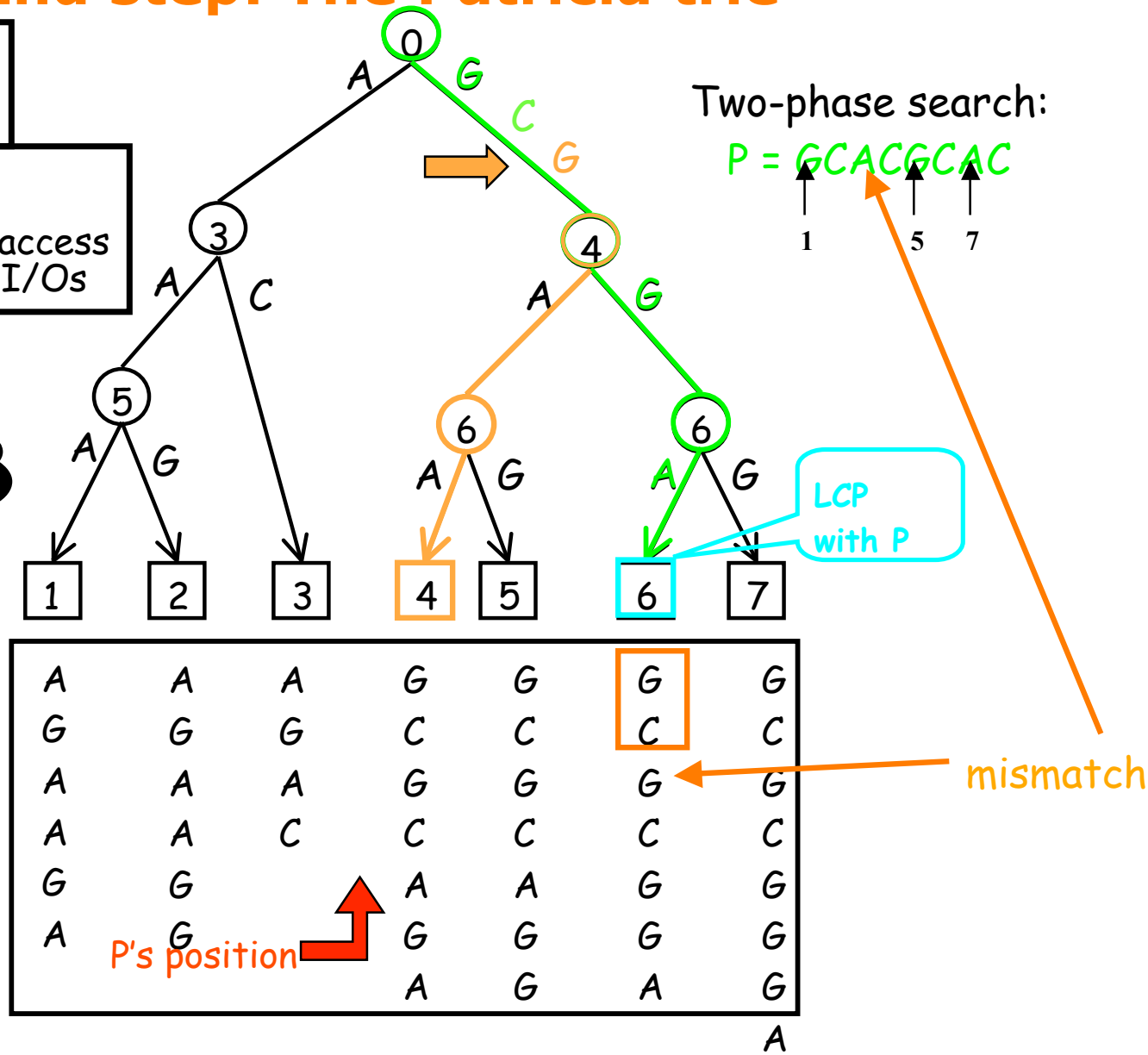


2nd step: The Patricia trie

Space PT
- O(k) , not O(N)

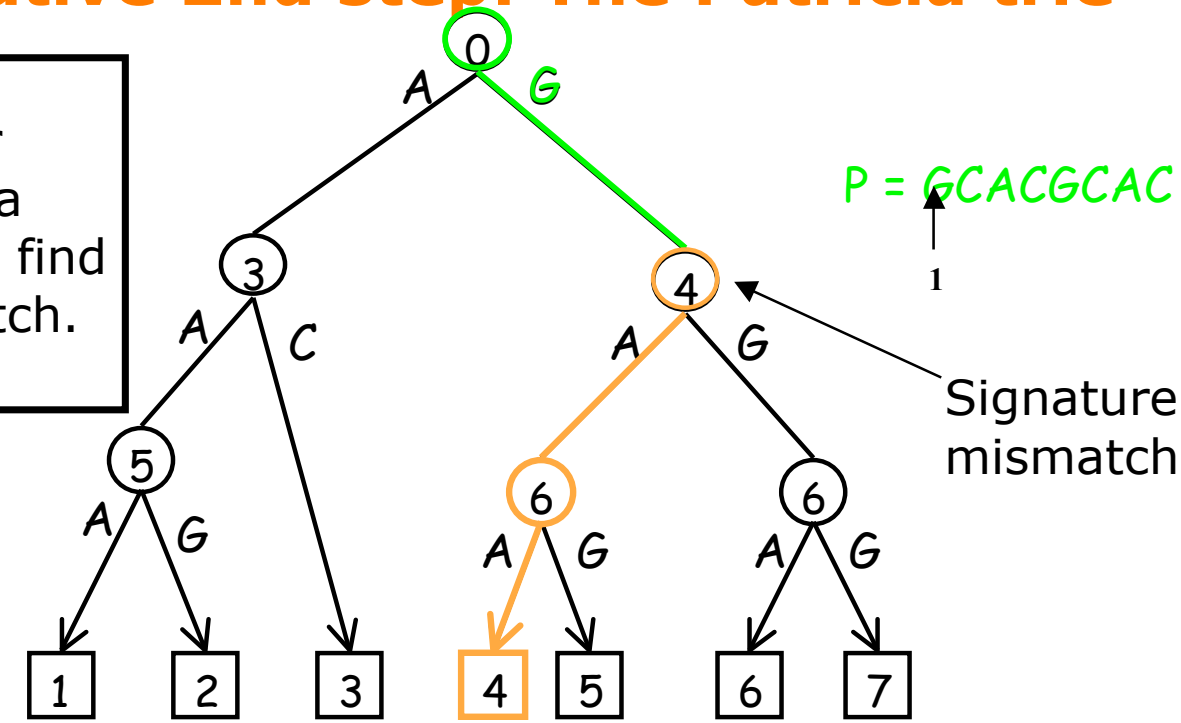
Search(P):
- First phase: no string access
- Second phase: O(p/B) I/Os

Just one string is checked !!



Alternative 2nd step: The Patricia trie

Store fingerprint for each node in Patricia trie. Can be used to find a longest prefix match.

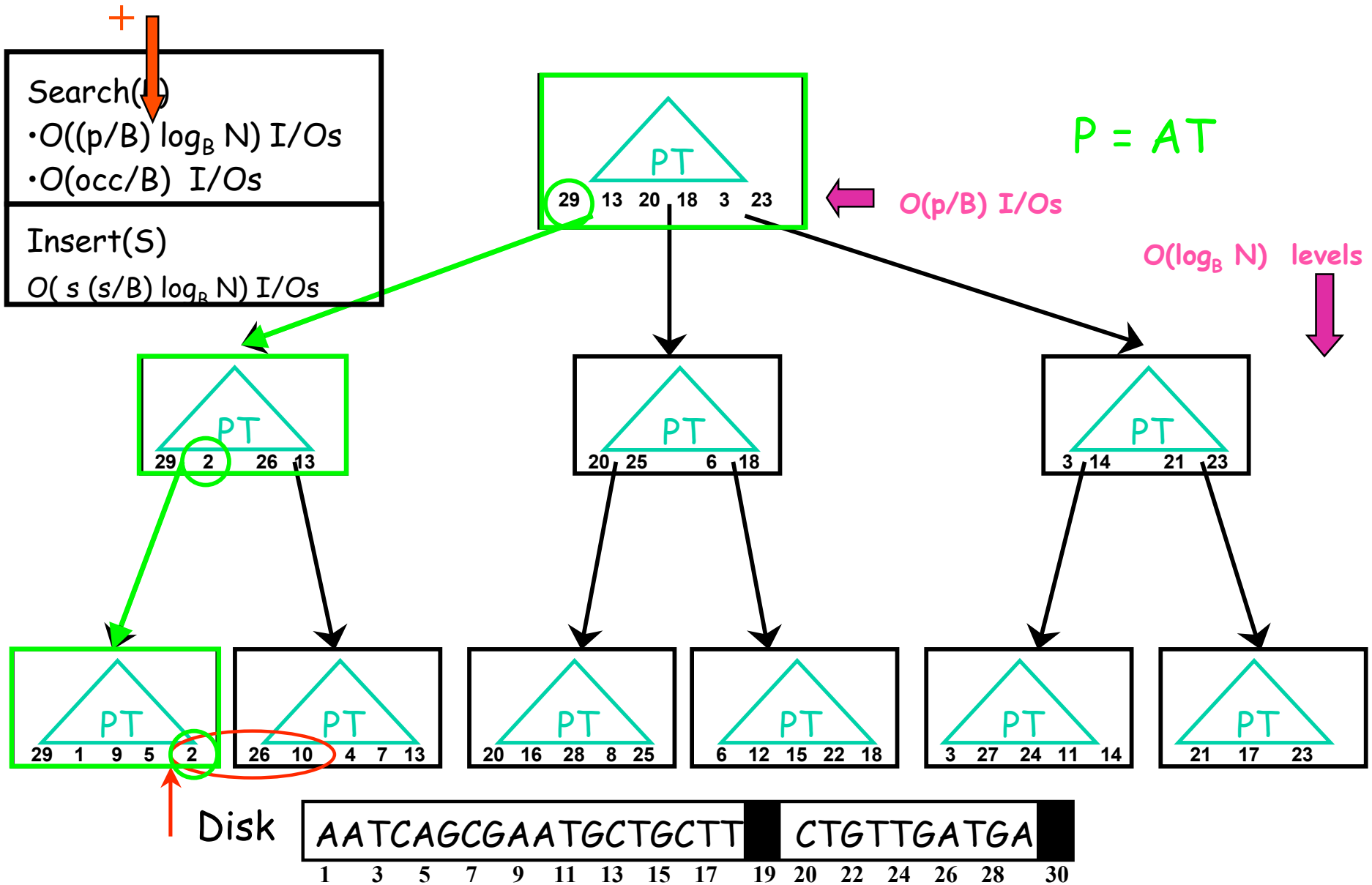


Disk

A	A	A	G	G	G	G
G	G	G	C	C	C	C
A	A	A	G	G	G	G
A	A	C	C	C	C	C
G	G		A	A	G	G
A	G		G	G	G	G
			A	G	A	G
						A

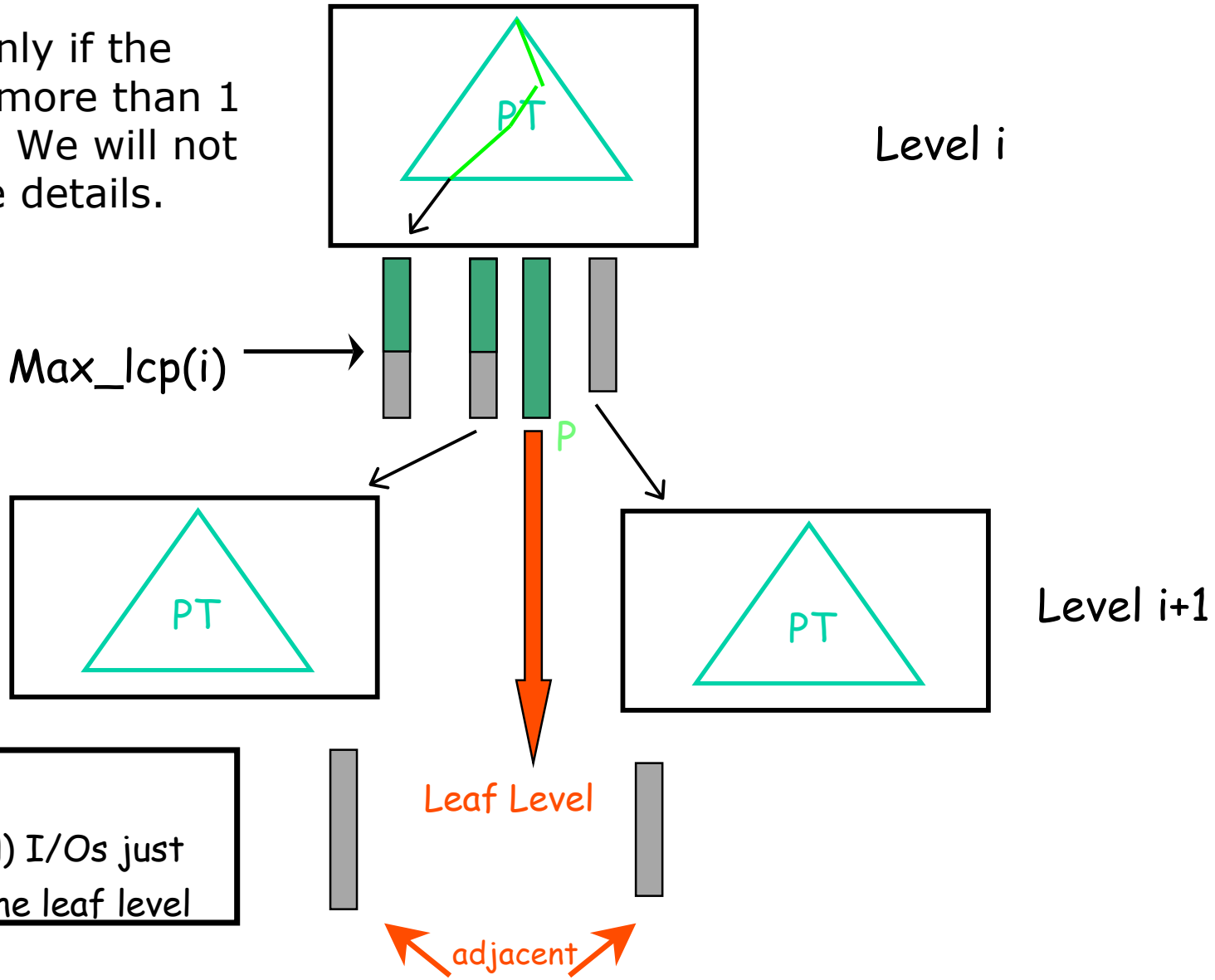
P's position

3rd step: B-tree + Patricia tree



4th step: Incremental Search

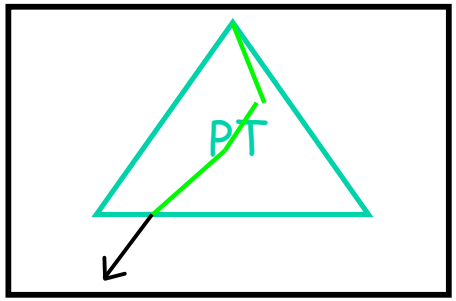
An issue only if the pattern is more than 1 disk block. We will not go into the details.



Search(P)
- $O(\log_B N)$ I/Os just to go to the leaf level

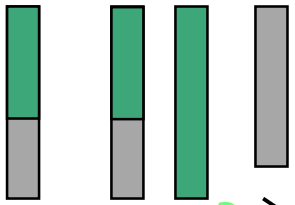
4th step: Incremental Search

Search(P)
- $O(p/B + \log_B N)$ I/Os
- $O(occ/B)$ I/Os



Level i

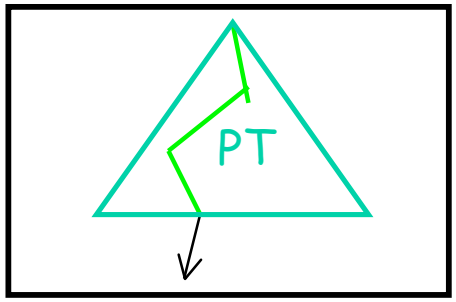
Max_lcp(i)



P

Inductive Step

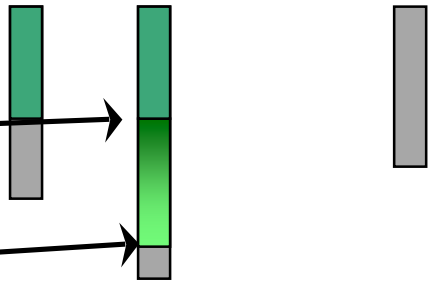
No rescanning



Level i+1

i-th step:
$O((lcp_{i+1} - lcp_i)/B + 1)$ I/Os

skip Max_lcp(i)



Max_lcp(i+1)

Summary: String B-tree

[Ferragina-Grossi, 95]

String B-tree performance:

- Search(P) takes $O(p/B + \log_B N + occ/B)$ I/Os
- Update(S) takes $O(s \log_B N)$ I/Os
  (more efficient buffered variant is possible)
- Space is $\Theta(N/B)$ disk pages

Crux: Search time independent of the length of the strings stored.

Using the String B-tree in internal memory:

- Search(P) takes $O(p + \log_2 N + occ)$ time
- Update(S) takes $O(s \log_2 N)$ time
- Space is $\Theta(N)$ bytes

➢ It is a sort of dynamic suffix array

Text indexing in Oracle

Main index types, CONTEXT and CTXCAT.

```
create index on mytable(attr)
  indextype is ctxsys.context;

create index on mytable(attr)
  indextype is ctxsys.ctxcat
```

To use, must use special string query language (not just a LIKE operator). (Why?)

Text indexing in Oracle

- Unlike ordinary indexes, `CONTEXT` indexes are NOT updated as content is changed. A ”refresh interval” can be specified.
- This means they are only useful when the tables are largely read-only and/or ”the end-users don’t mind not having 100% search recall”.
- `CTXCAT` is transactional (i.e. kept updated) - but indexes each row separately.
- Unfortunately, I was not able to find information about the implementation.

Text indexing in Oracle

From the documentation:

- **CTXCAT Indexes** - A CTXCAT index is best for smaller text fragments that must be indexed along with other standard relational data (VARCHAR2).

  ```
  WHERE CATSEARCH(text_column, 'ipod')> 0;
  ```

- **CONTEXT Indexes** - The CONTEXT index type is used to index large amounts of text such as Word, PDF, XML, HTML or plain text documents.

  ```
  WHERE CONTAINS(test_column, 'ipod', 1) > 0
  ```