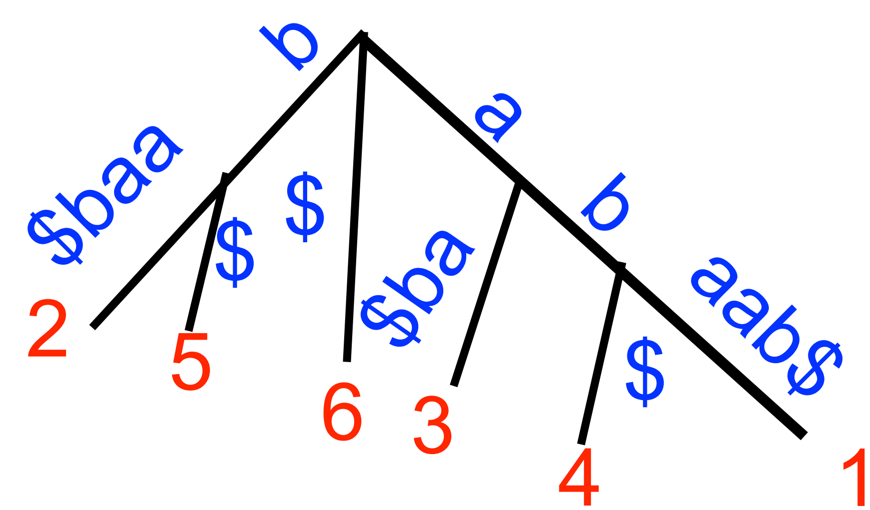
Suffix trees

Construction algorithm

McCreight's suffix tree construction algorithm



Motivation

Recall: the suffix tree is an extremely useful data structure with space usage and construction time in $O(n)$.

Today we see an algorithm for constructing a suffix tree in time $O(n)$.

Suffix trees

A suffix tree of a sequence, *x*, is a *compressed/ compacted trie* of all suffixes of the sequence *x*$.

x=abaab

0: abaab$

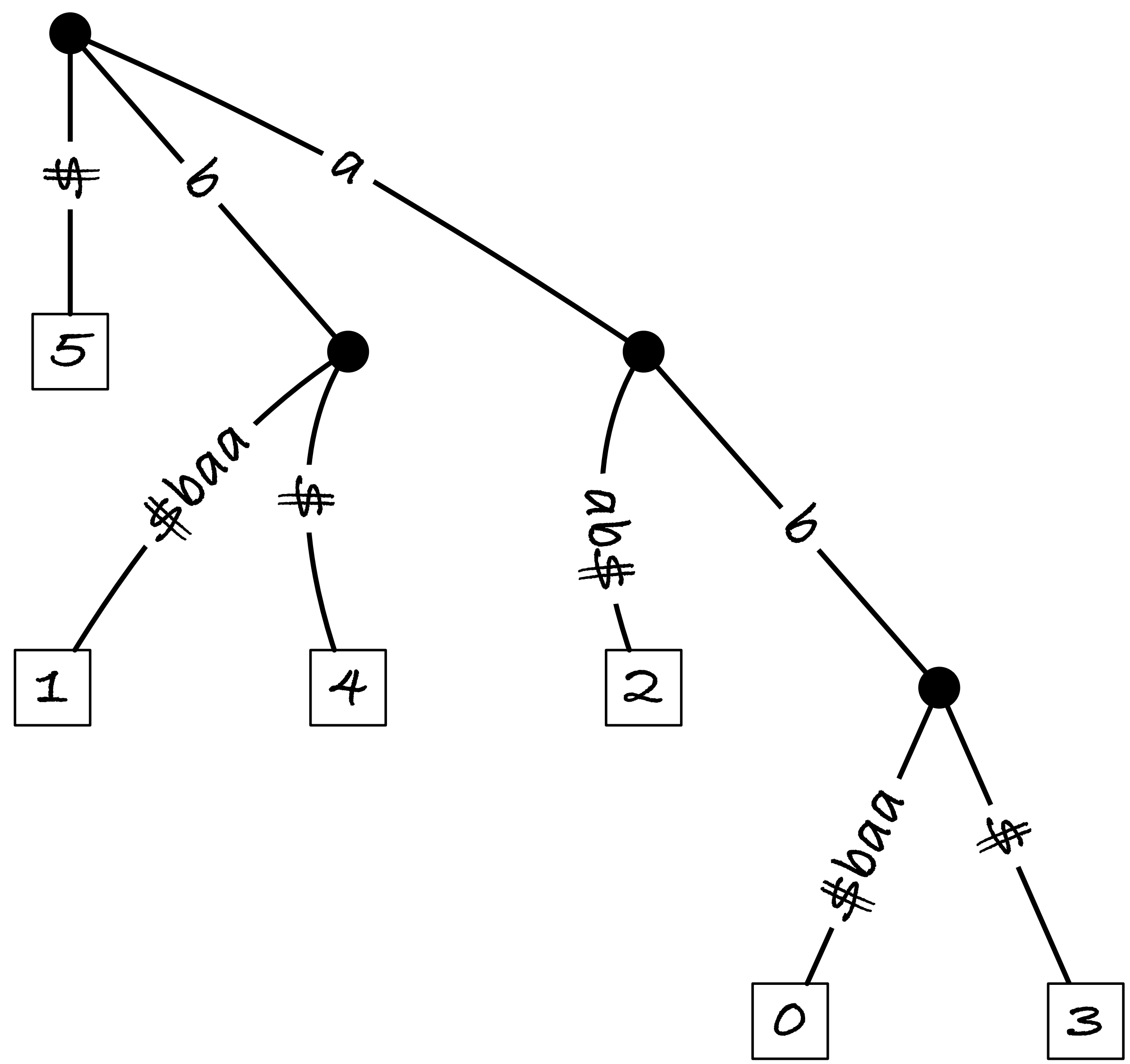
1: baab$

2: aab$

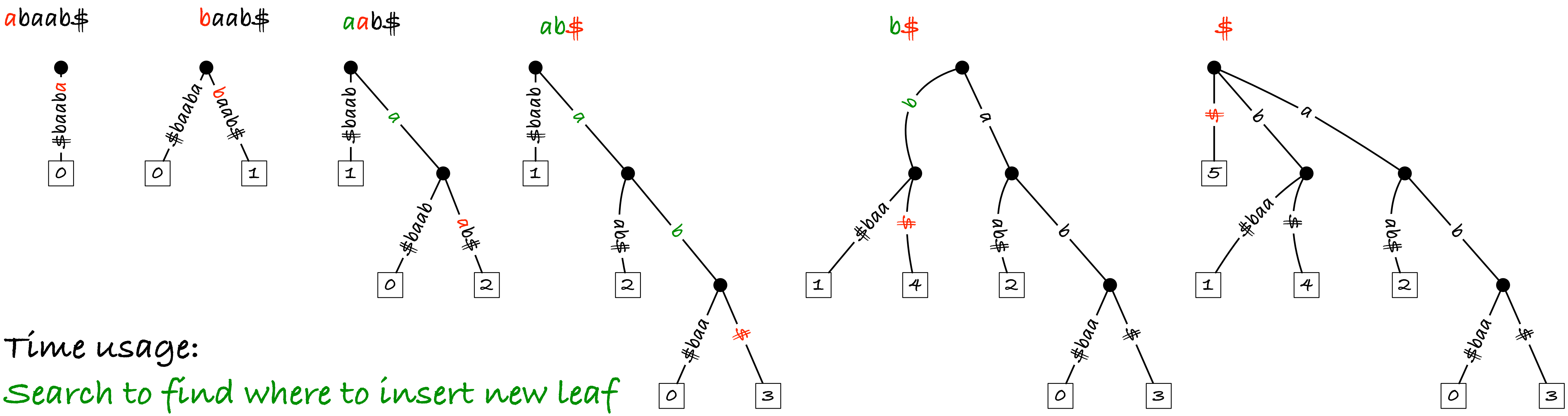
3: ab$

4: b$

5: $



Naïve construction



Time usage:

Search to find where to insert new leaf

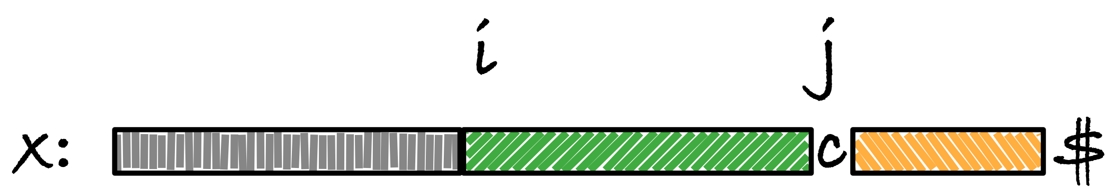
Insert a leaf in the tree

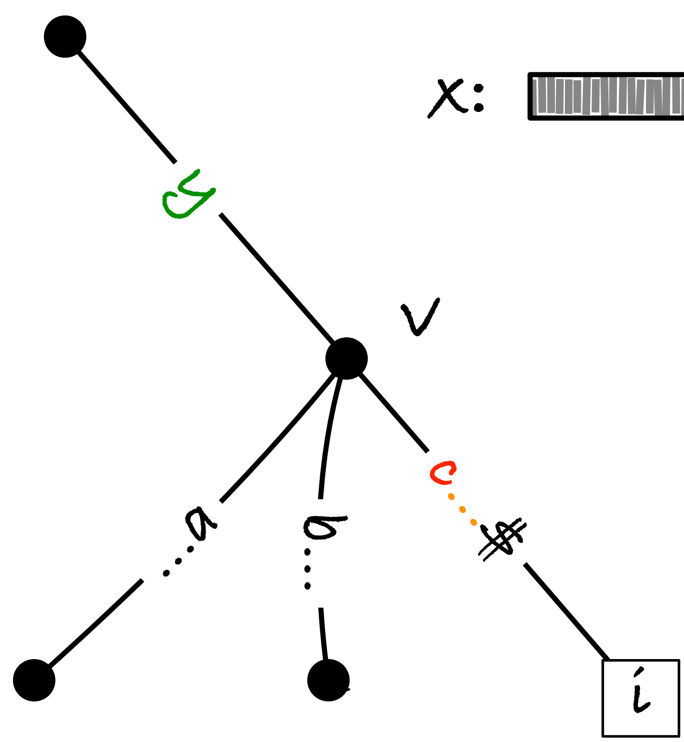
Insert is a little pointer manipulation

O(1) per suffix

The troublesome part is searching

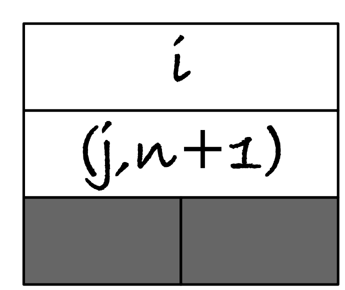
Add new leaf



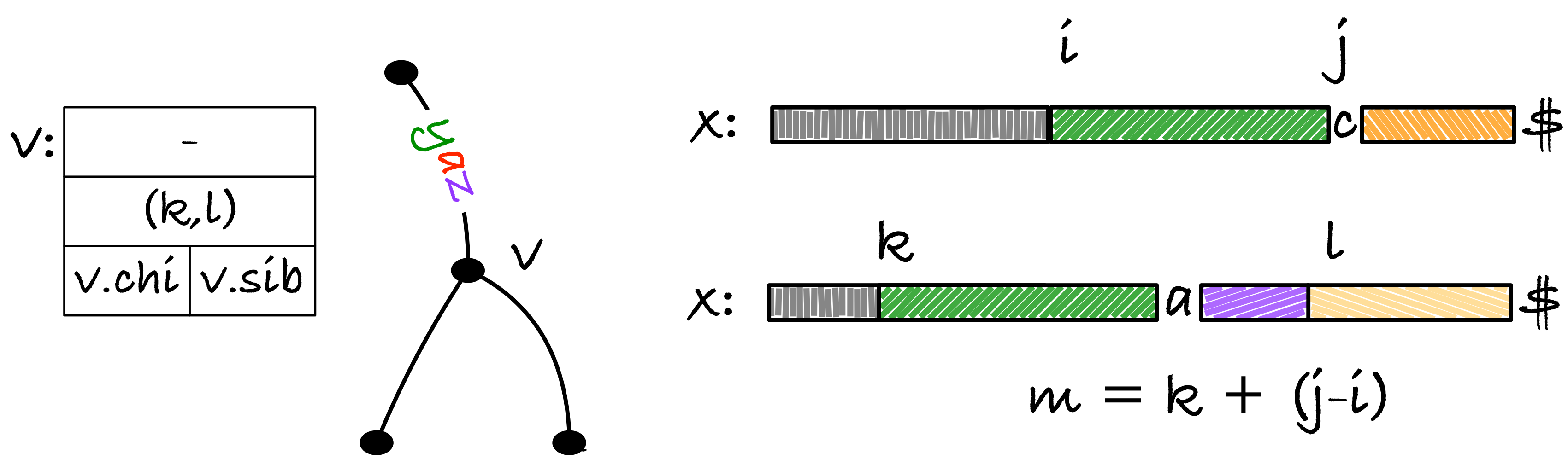


Insert on node:

create new leaf node and connect to v



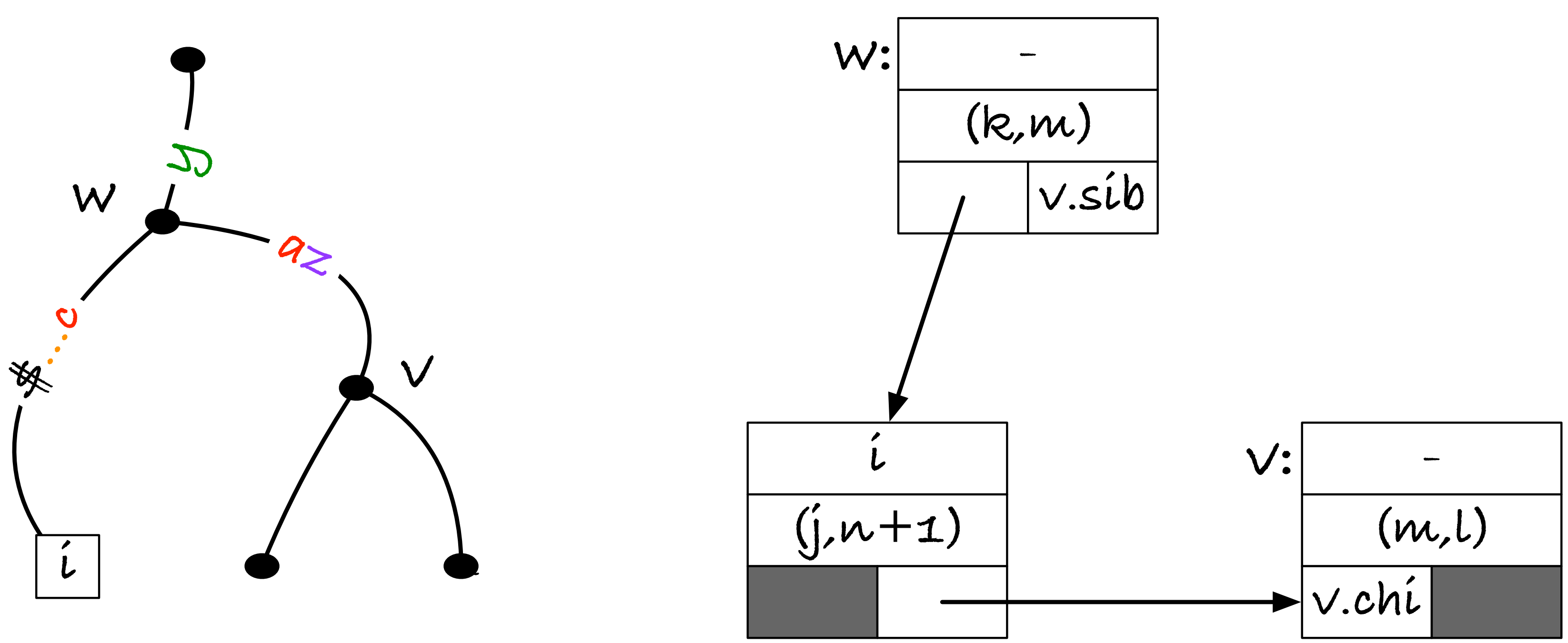
Add new leaf



$m = k + (j-i)$

Split edge:

create new leaf node and new node to split edge. Update edge intervals and connect pointers

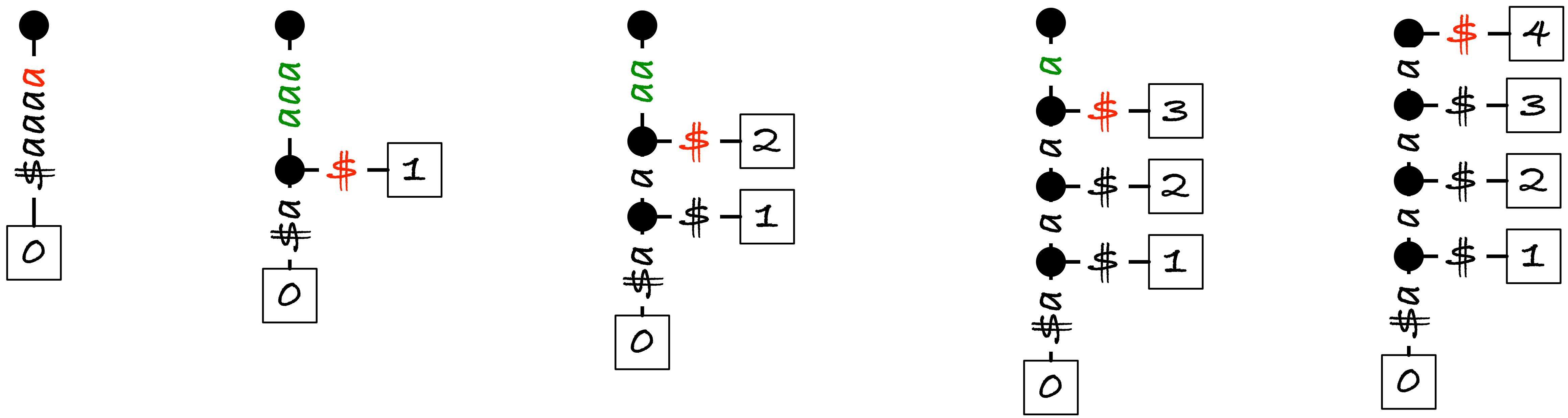


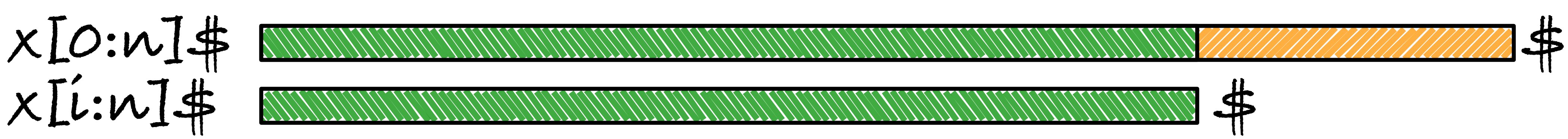
Running time

- How much time do we spend on searching for the break point?

- What could a worst-case string look like?

Running time





Suffix $0 < i \leq n$ matches for $n - i$ characters.

Thus $O(n^2)$ total matches.

McCreight's algorithm

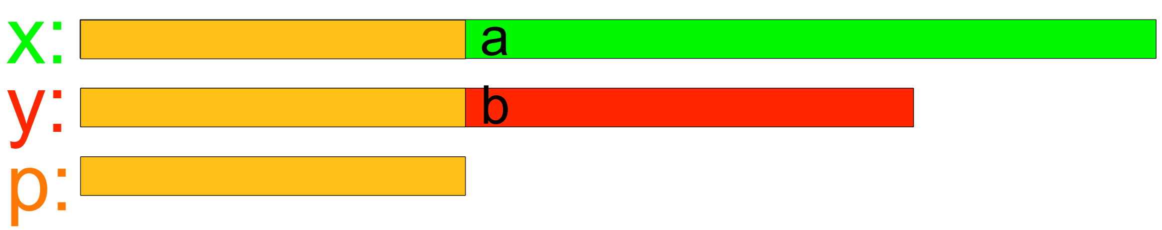
- McCreight’s algorithm builds a suffix tree by iteratively adding suffixes i=0, …, n to a compacted trie

- We get a linear running time by using two tricks to search faster

Terminology

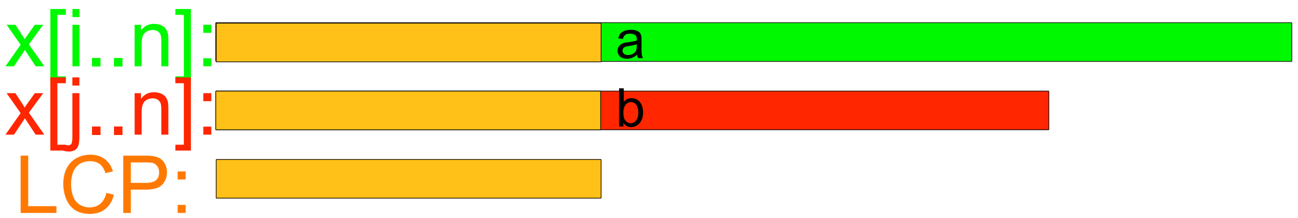
- A *common prefix* of x and y is a string, p, that is a prefix of both:

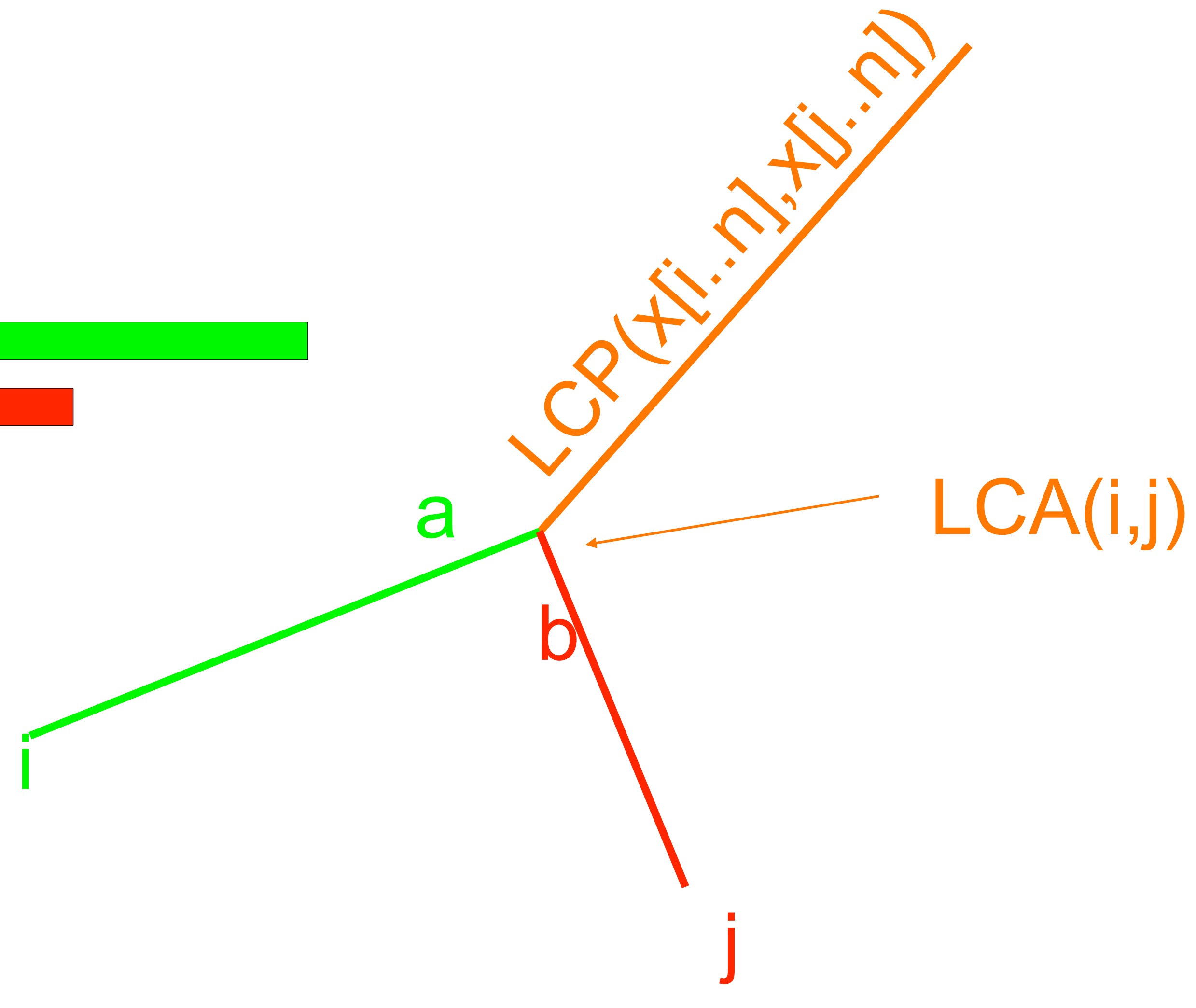
- The *longest* common prefix, p=LCP(x,y), is a prefix such that: $x[|p|+1] \neq y[|p|+1]$



LCP and LCA

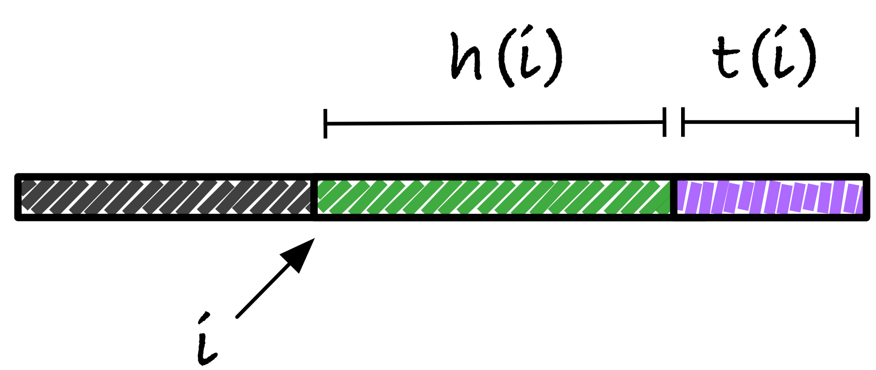
- For suffixes of x, x[i..n], x[j..n], their longest common prefix is their *lowest common ancestor* in the suffix tree:

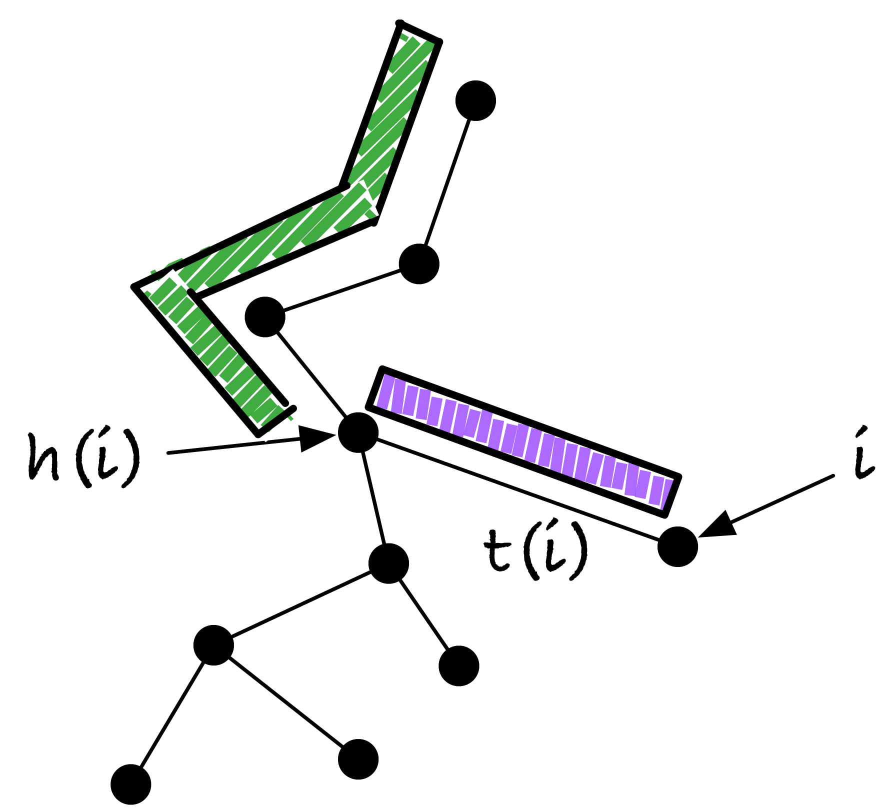




Head and tail

- Let head(i) denote the longest LCP of $x[i..n]\$$ and $x[j..n]\$$ for all $j < i$ (it's the longest string we can match)
- Let tail(i) be the string such that $x[i..n]\$$=head(i)tail(i)
- Iteration i in McCreight's algorithm consist of
  - finding (or inserting) the node for head(i),
  - and appending tail(i)



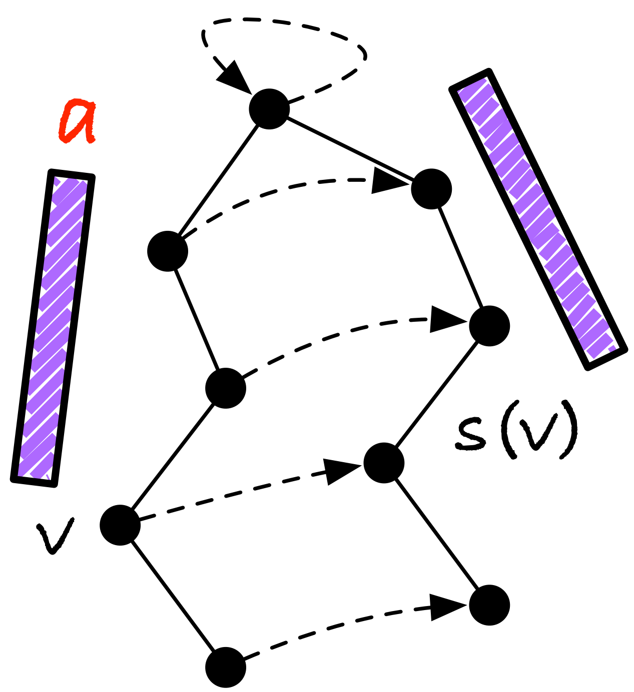


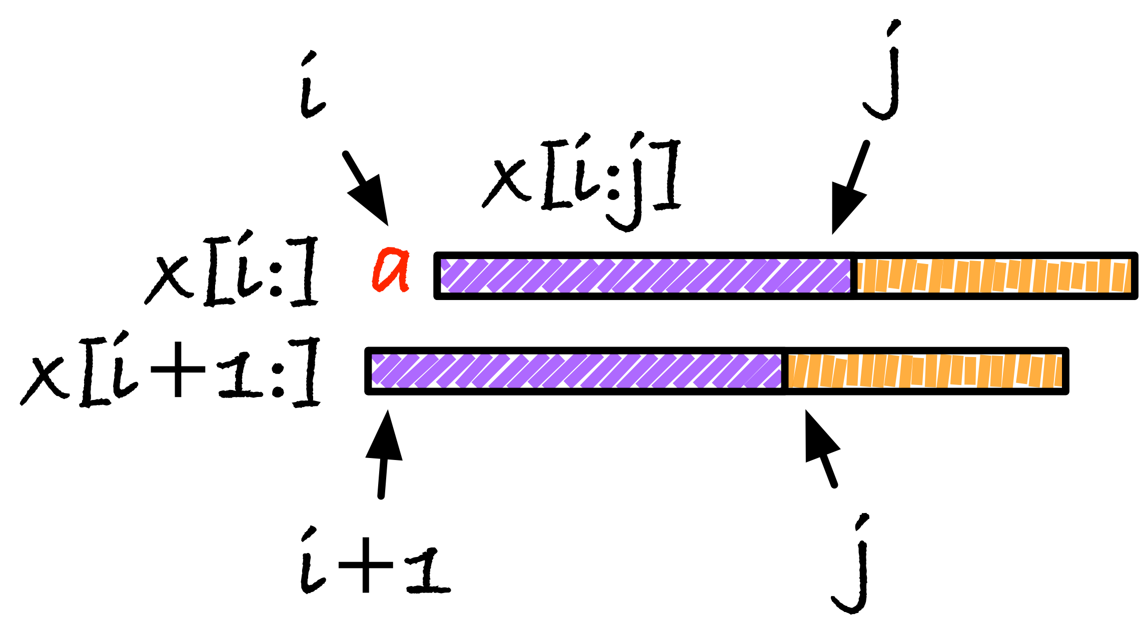
Head and tail

- *Head* is just the name we give for the piece of string we need to search down the current trie
- We define it to be a string that is in the trie, *but we don't know what it is or where it is!*
  - If we knew head(i), things would be easier, but we only know that it exists
  - There is a difference between know that a specific (known) string is in a trie and knowing that some (unknown) string must be there. We can do more with the former (as we shall see).

Suffix link

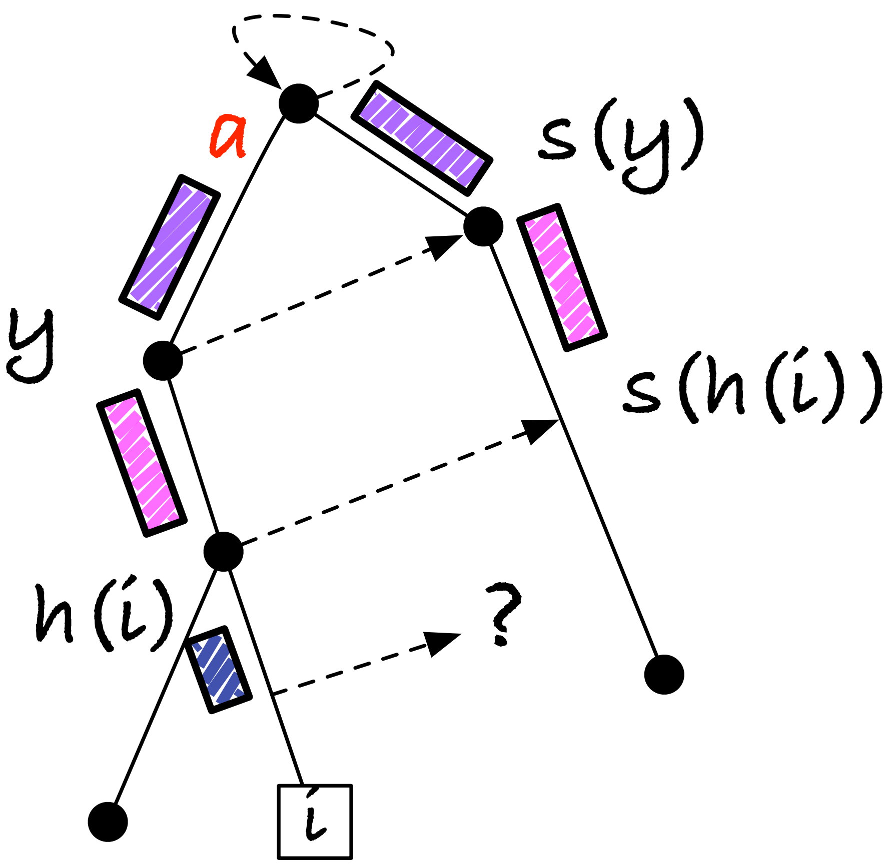
- Define s(u) = “” if u=””, and s(u)=v if u=av
  - Same as for Aho-Corasick since all our strings are suffixes of x.

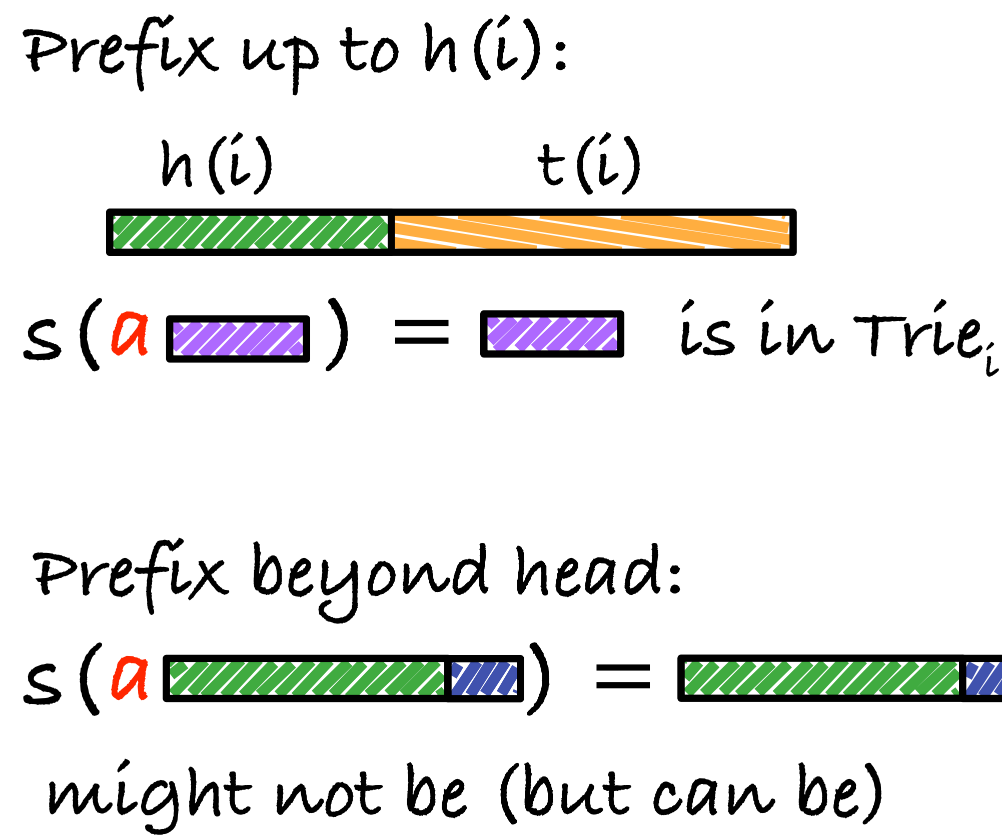




Suffix link

- We need the full suffix tree before we have all suffixes in the trie… along the way, we only have some

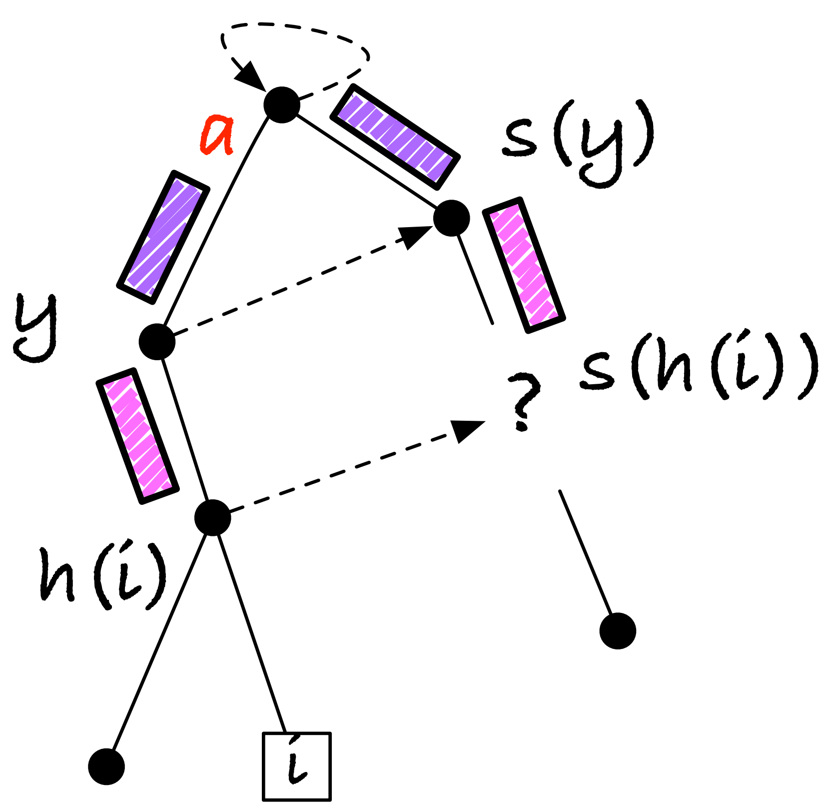




(we show this property in the exercises; not essential for the algorithm)

Suffix link

- We can't have pointers on the edges (why?) so we only have suffix links on nodes.
  - We do not quite have the nodes we need, but good enough
  - This is the important property for the algorithm!



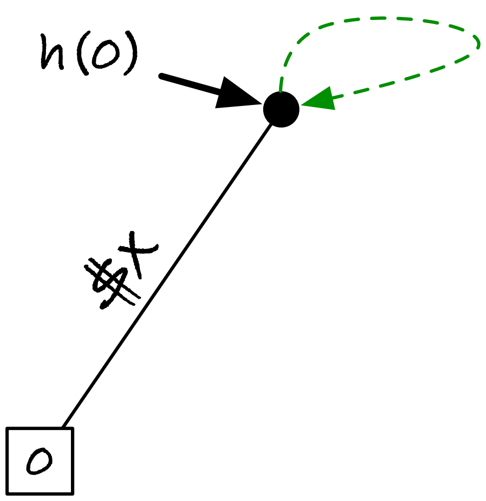
After inserting suffix i, s(h(i))
is in the trie, but it might not be a node
so we cannot have a pointer to it.

For all other inner nodes v, s(v)
is a node that we can add a pointer to.
(exercise)

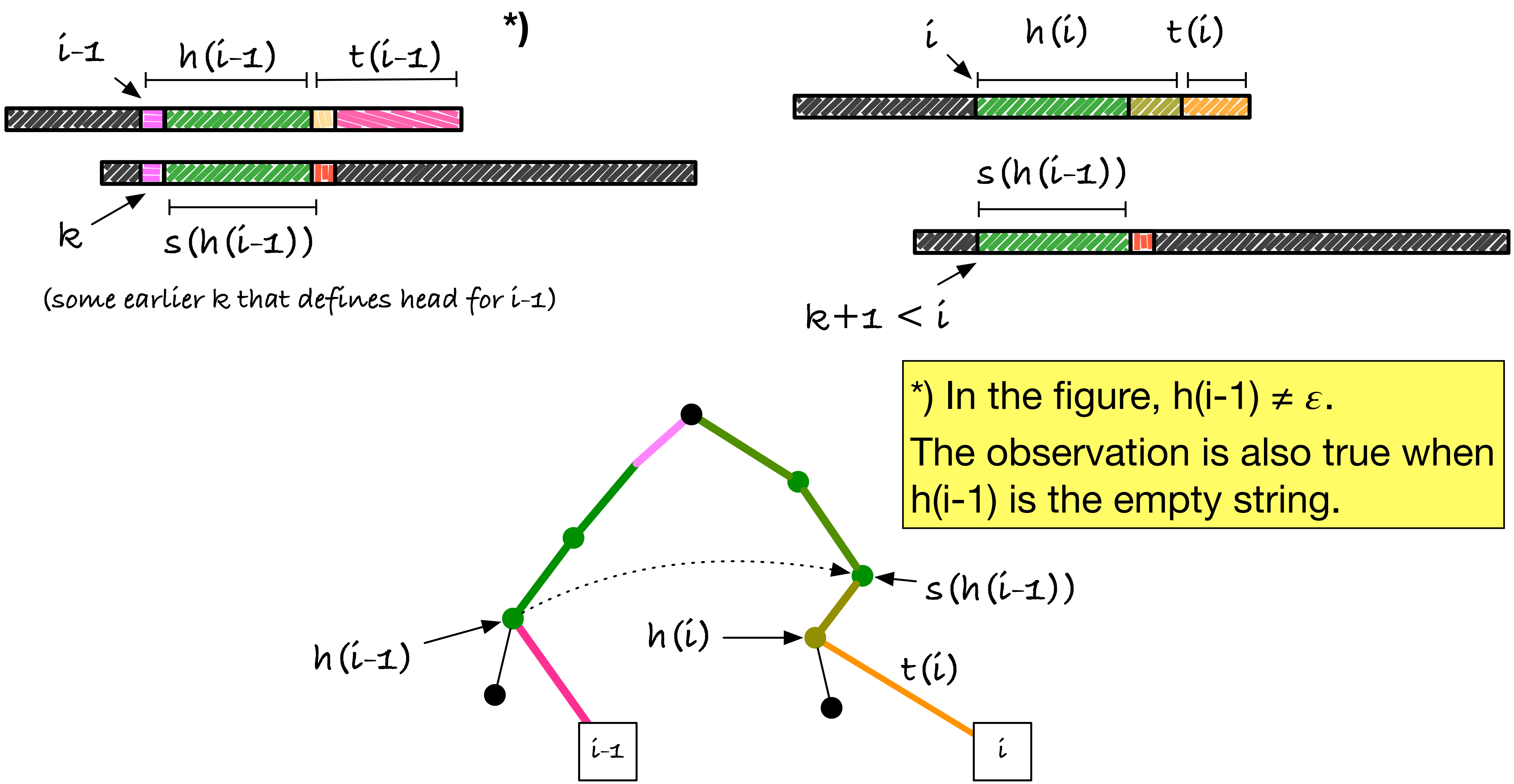
We also show the property in the proof for the algorithm.
We explicitly set the pointers, demonstrating that it is possible.

McCreight plan

- Start by creating the root and insert the first suffix.
    - Set the root's suffix link to the root.
- Iteratively i=1,…,n, insert suffix x[i:]$
    - When we insert suffix x[i:]$ we have h(i-1) (we just found it, so just don't forget it; h(0) is the root).
    - h(i-1) is the starting point for our search for h(i)



Head and suffix links

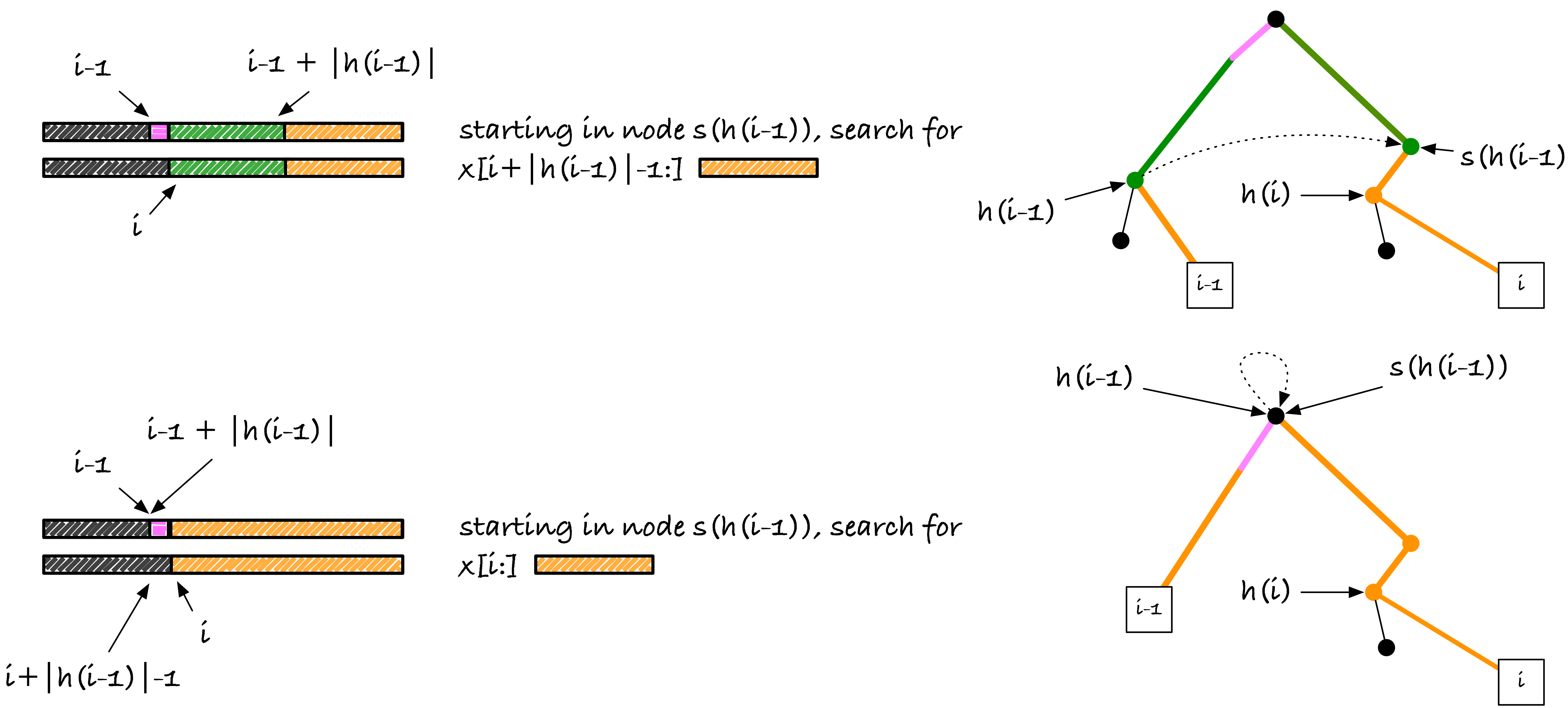


*) In the figure, h(i-1) $\neq \varepsilon$.

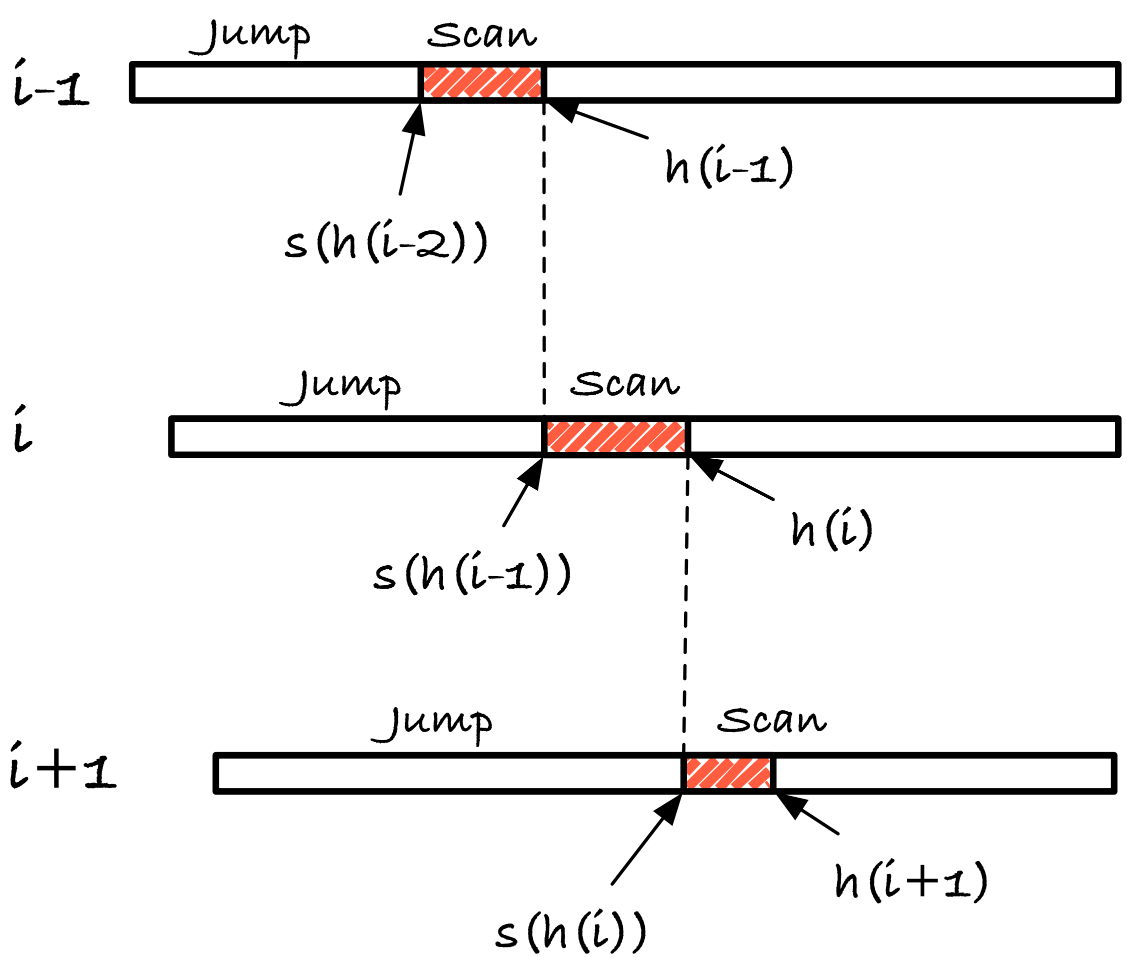
The observation is also true when h(i-1) is the empty string.

Head and suffix links

The property is true for both empty and non-empty heads, but when you implement it, you have to consider the empty head a special case if you want to know what you search for…

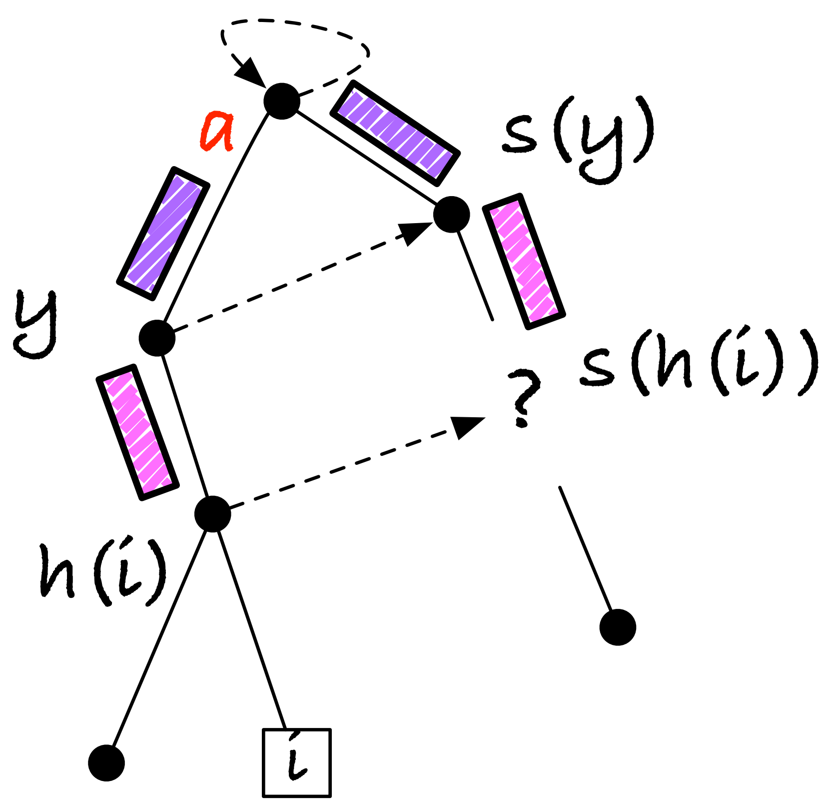


Head and suffix links



If we jump with suffix links, the scans are non-overlapping. The total scan time is O(n)!

Not so fast!



~~After inserting suffix i, s(h(i))~~

$\boxed{\text{is in the trie, but it might not be a node so we cannot have a pointer to it.}}$

For all other inner nodes v, s(v)
is a node that we can add a pointer to.
(exercise)

All inner nodes **except the one we want** have a suffix link

(shit)

We need to do a little more work after all

Also, we didn't actually set any suffix links, and no-one does it for us, so even without this crappy result we are not done.

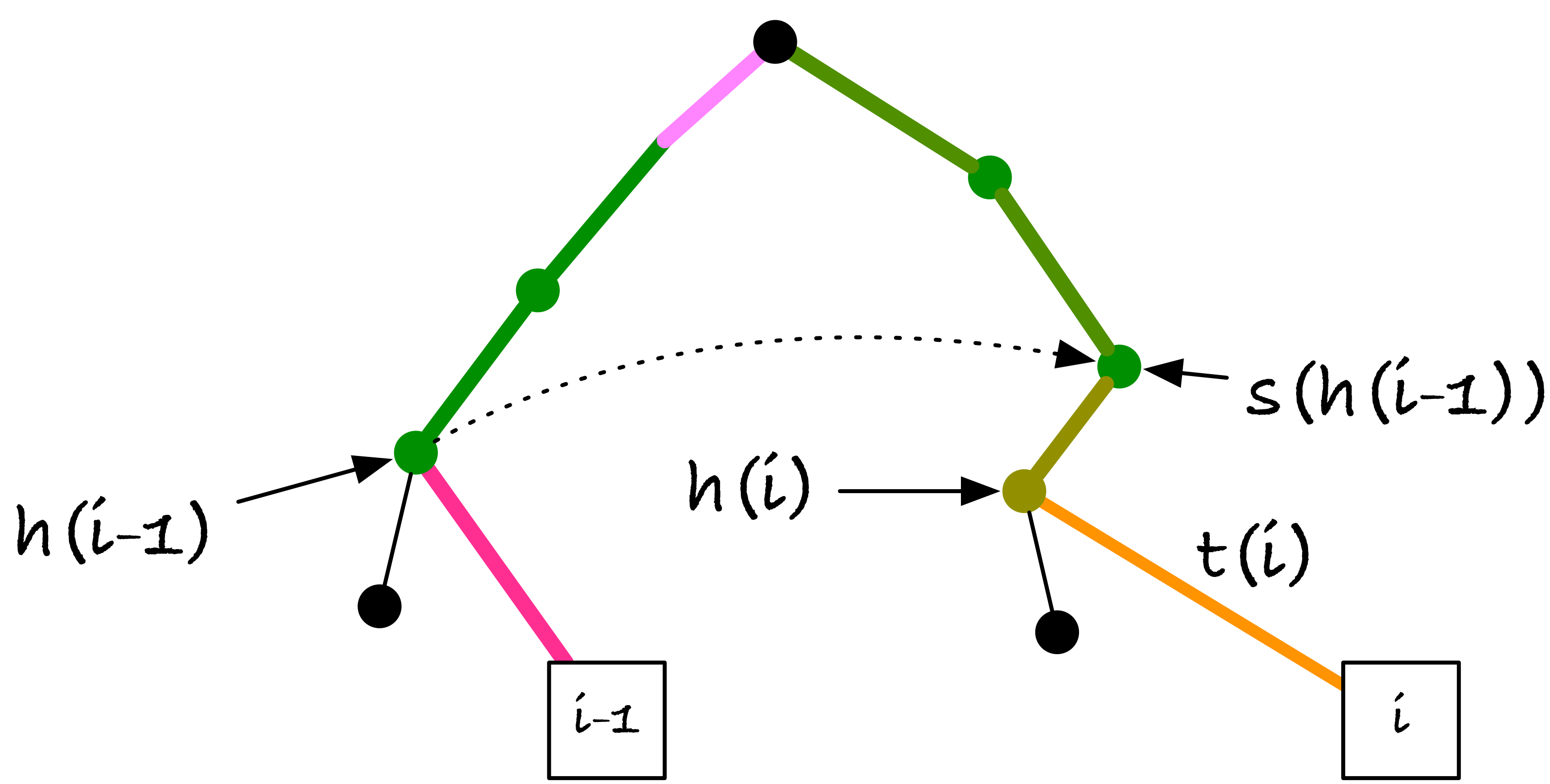
Setting suffix links

Every time you search for h(i) you need to go past s(h(i-1))
[unless we find a way to jump past, which we will be careful not to].

When you do, set the pointer from h(i-1) to s(h(i-1)).

When you set it, it is the only node that doesn't have a suffix link,
so when you set it before you create h(i), h(i) will be the only one by then.

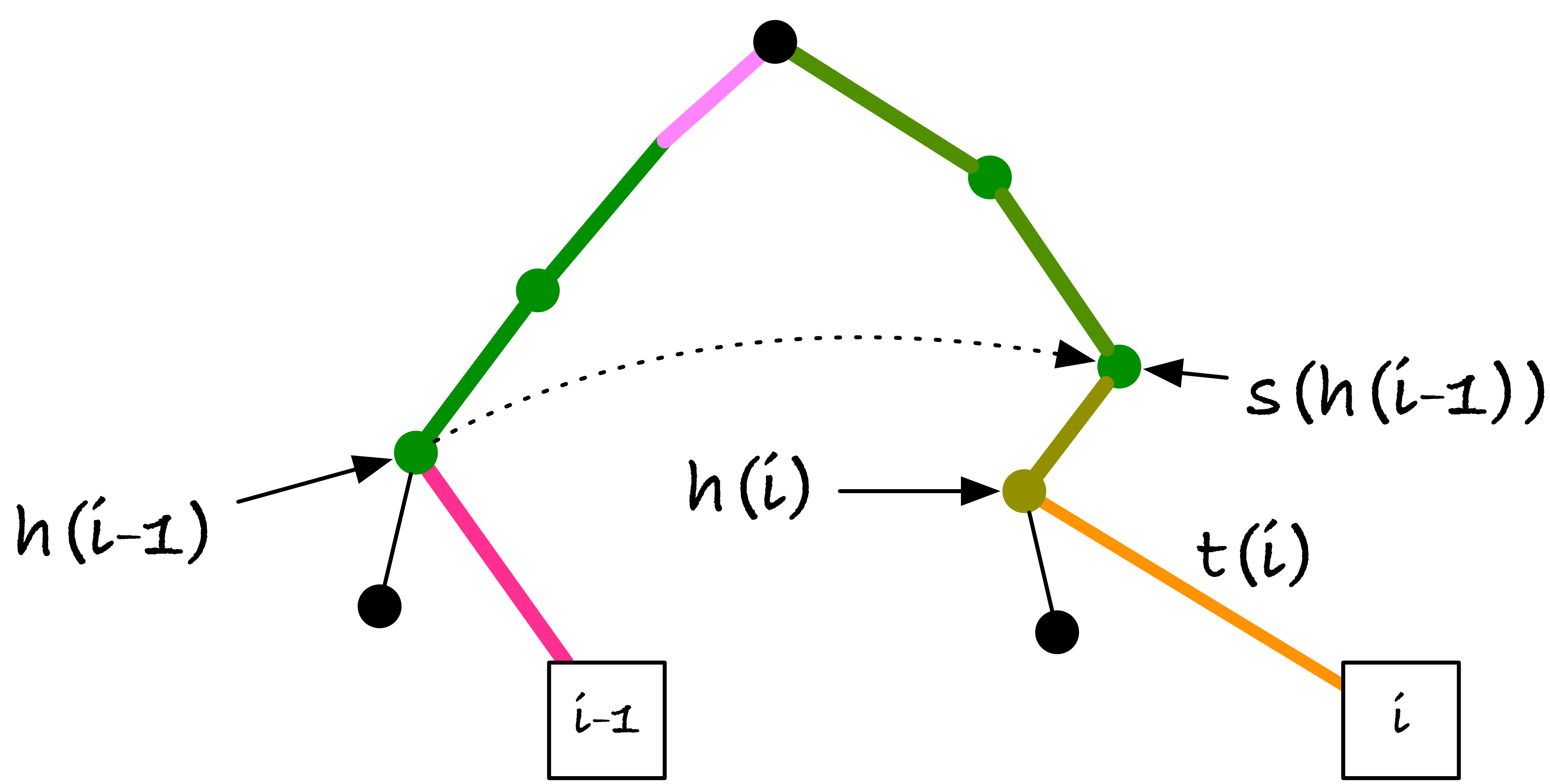
That was the easy fix.



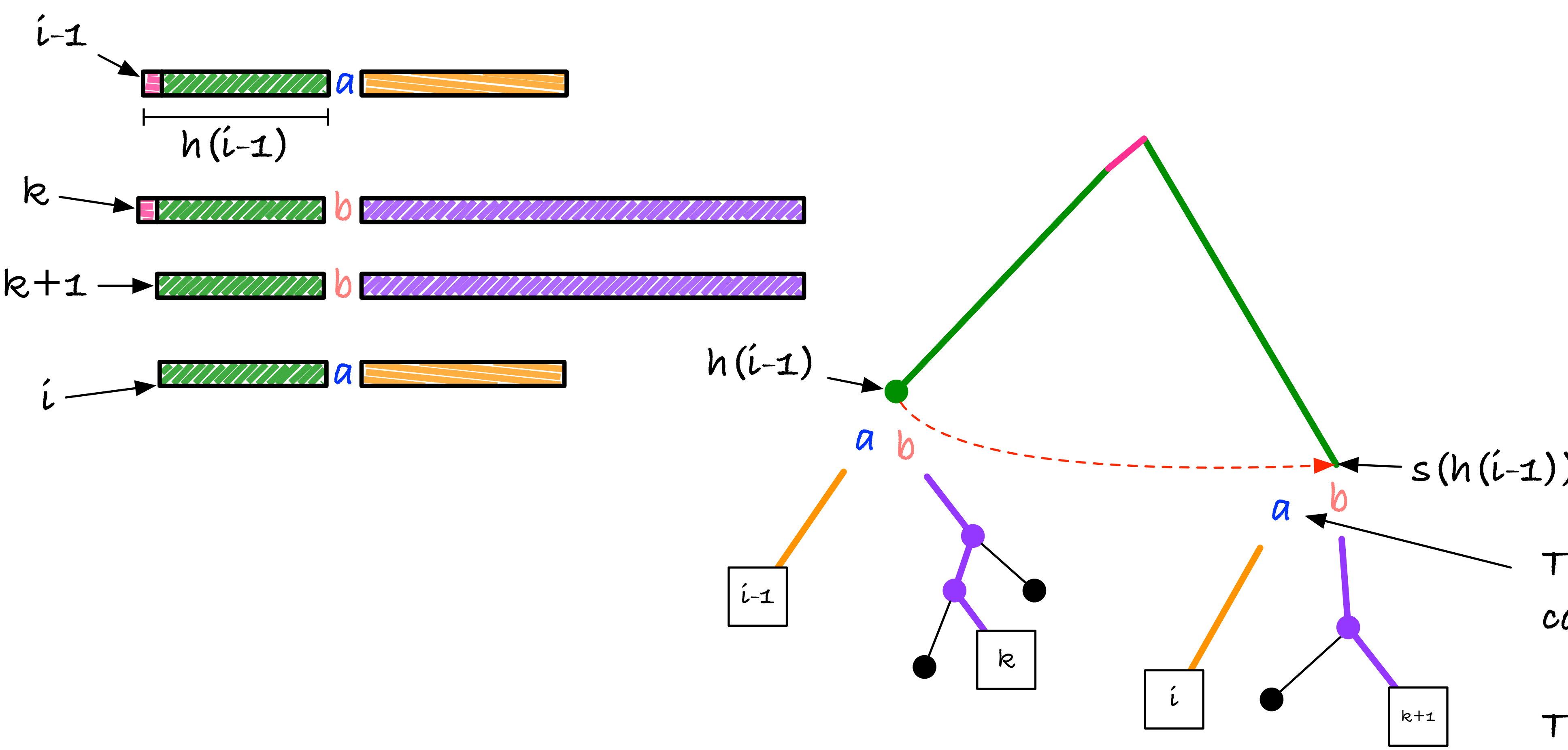
Setting suffix links

Wait! What if s(h(i-1)) isn't a node when we reach it?

We can't set a pointer to a location on an edge!



Setting suffix links



This "a" doesn't match the only continuation ("b").

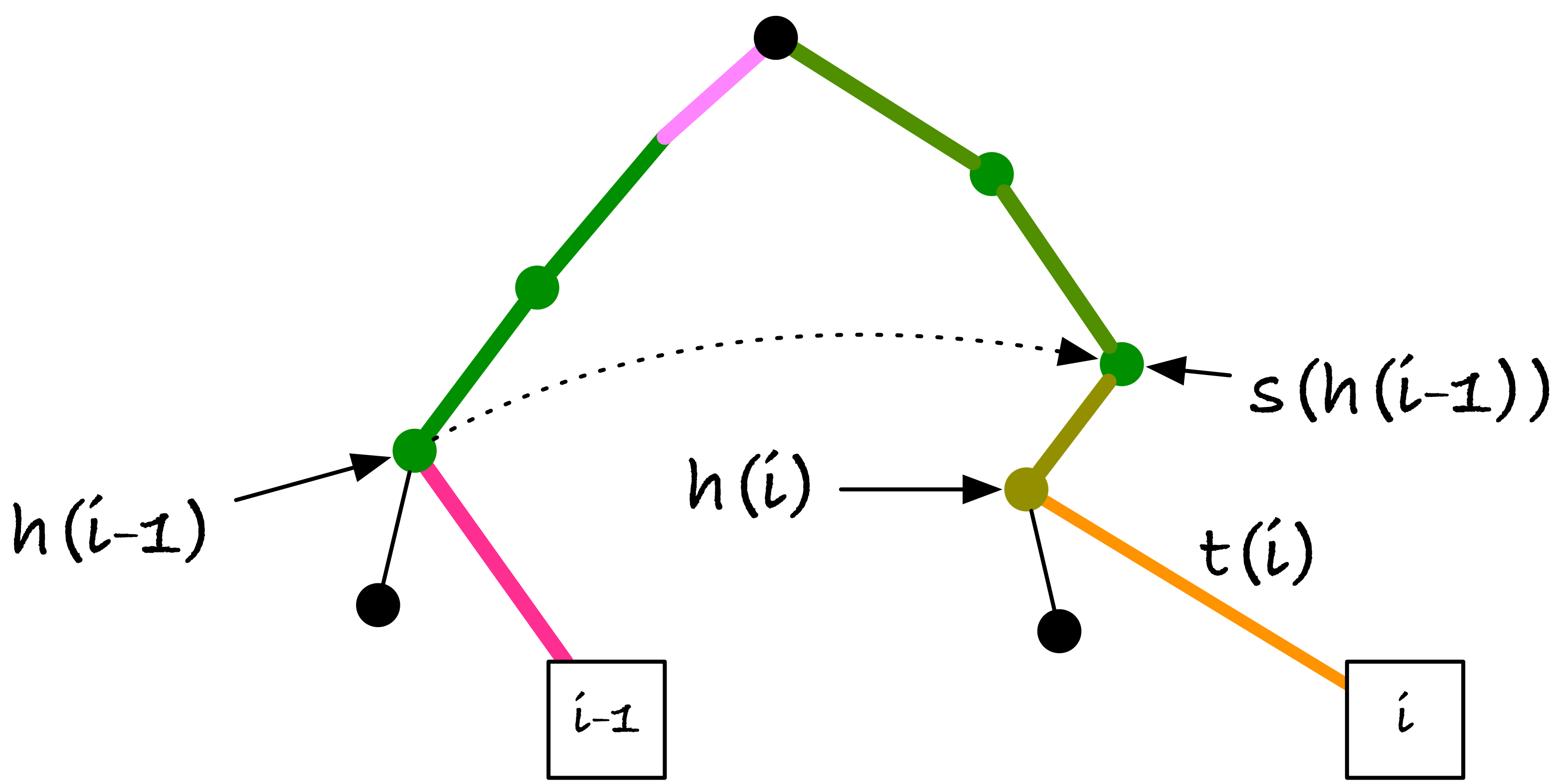
Thus, s(h(i-1)) must be h(i).

Insert it and we have s(h(i-1)).

Exercise: In one of the exercises you should give an example where this situation occurs.

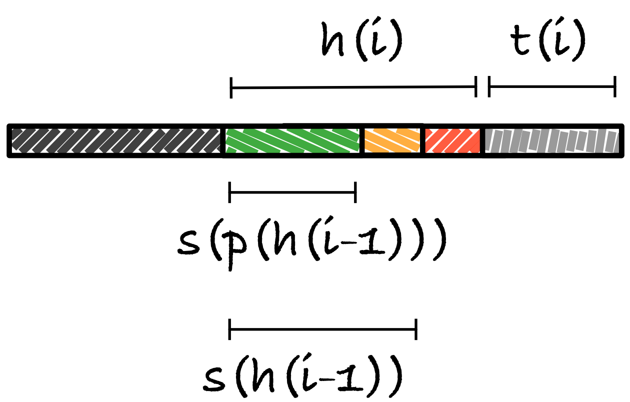
OK

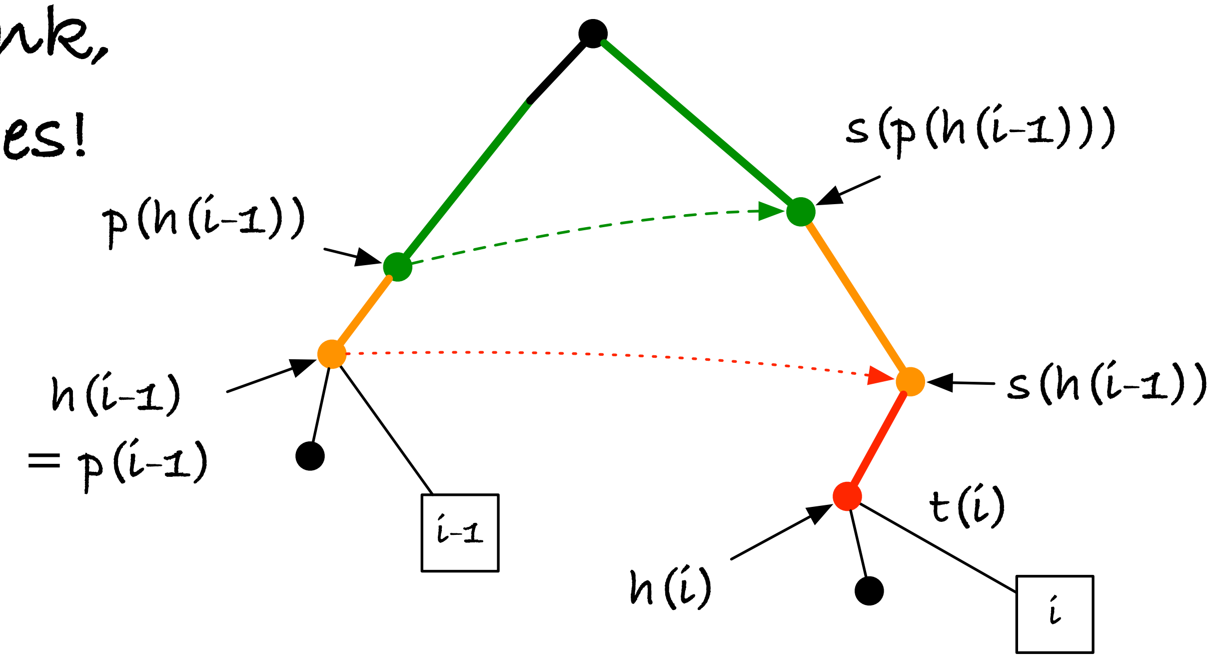
Suffix link pointers are taken care of, but how do we get from h(i-1) to s(h(i-1)) if we don't have the pointer?

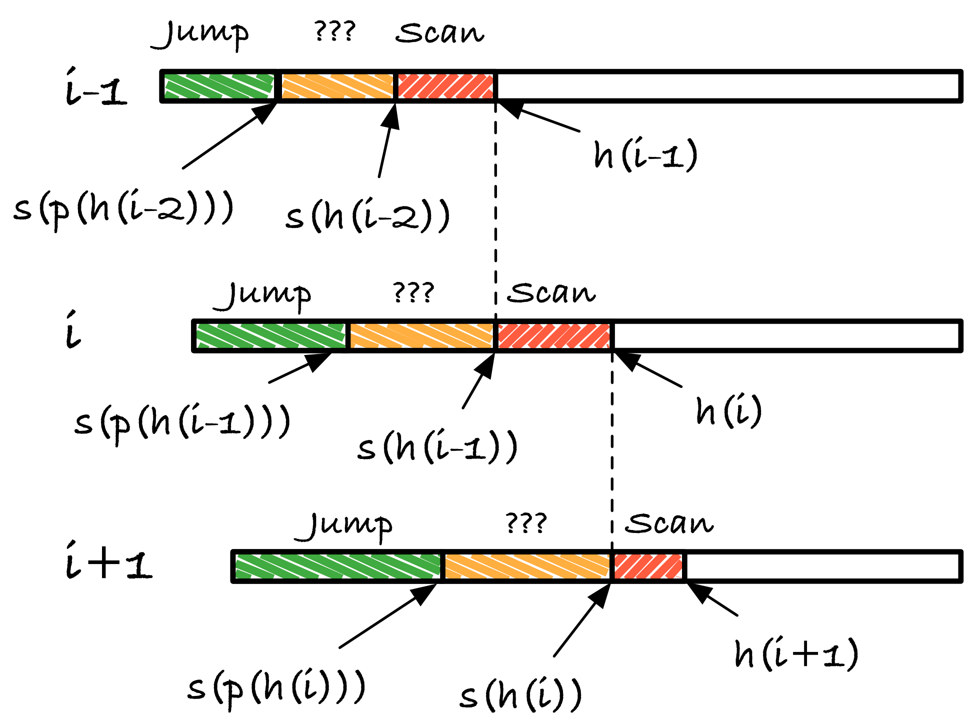


Parent of h(i-1)

h(i-1) might not have a link,
but its parent p(h(i-1)) does!







Jumps add up to O(n)

Scans add up to O(n)

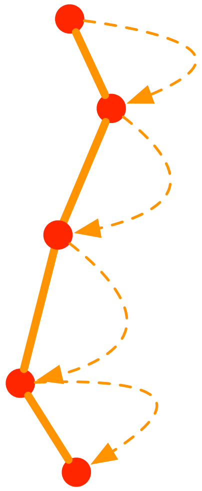
What about ???

NB! You need to add a parent pointer to go from node v to p(v) in O(1)

For convenience, you can make the parent of the root the root itself

Fast scan

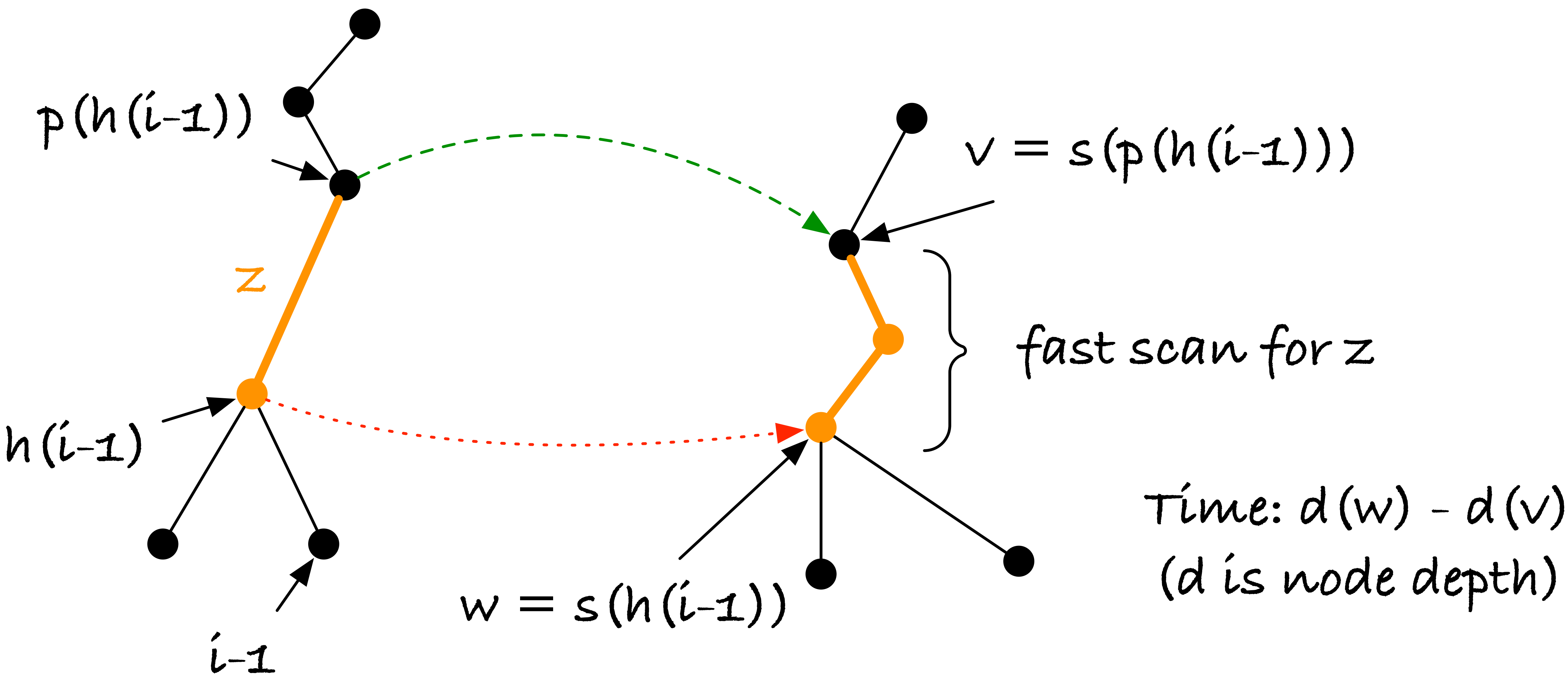
If you know that z is in the trie,
you can find where it is by jumping from node to node.
You only need to check z where you need to locate an out-edge.
The time is the number of nodes you pass through.



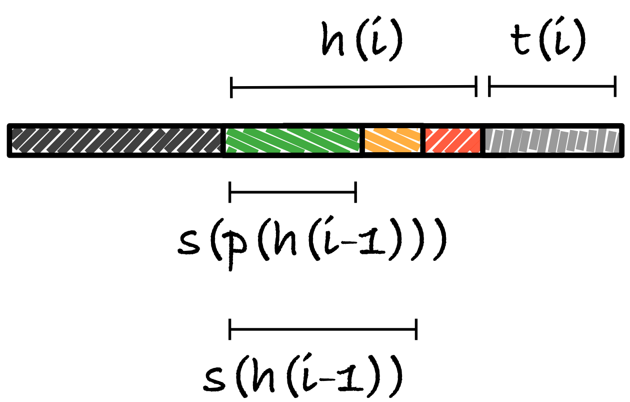
NB! We can't use this for h(i). We know that *a* h(i) exists, but we don't know what it is (we don't know its length). That is not enough. We need to know the actual string (or we won't know when to stop searching).

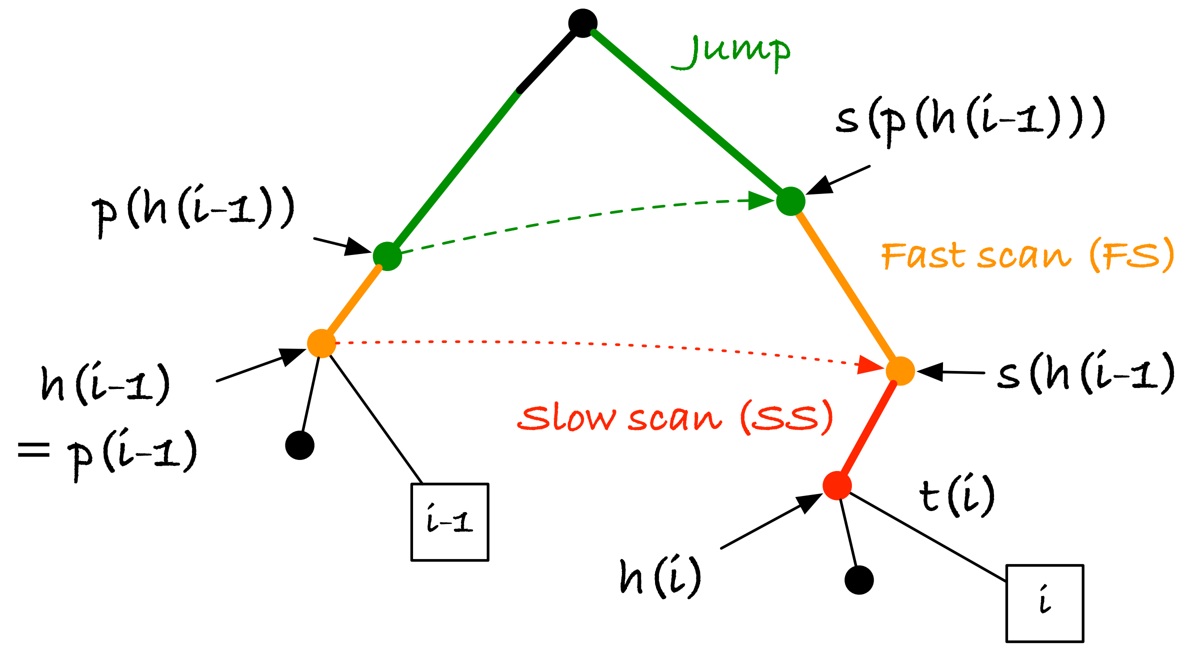
We do, however, know exactly what the string is between p(h(i-1)) and h(i-1), since it is right there on the edge.

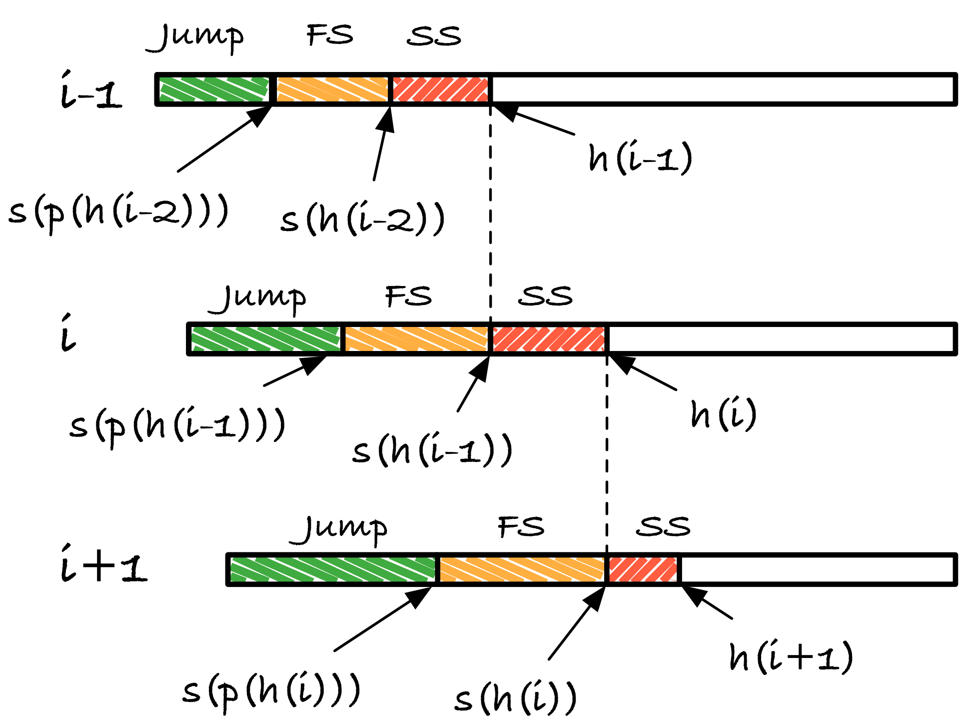
Fast scan


p(h(i-1))
v = s(p(h(i-1)))
z
fast scan for z
h(i-1)
Time: d(w) - d(v)
(d is node depth)
w = s(h(i-1))
i-1

McCreight's algorithm





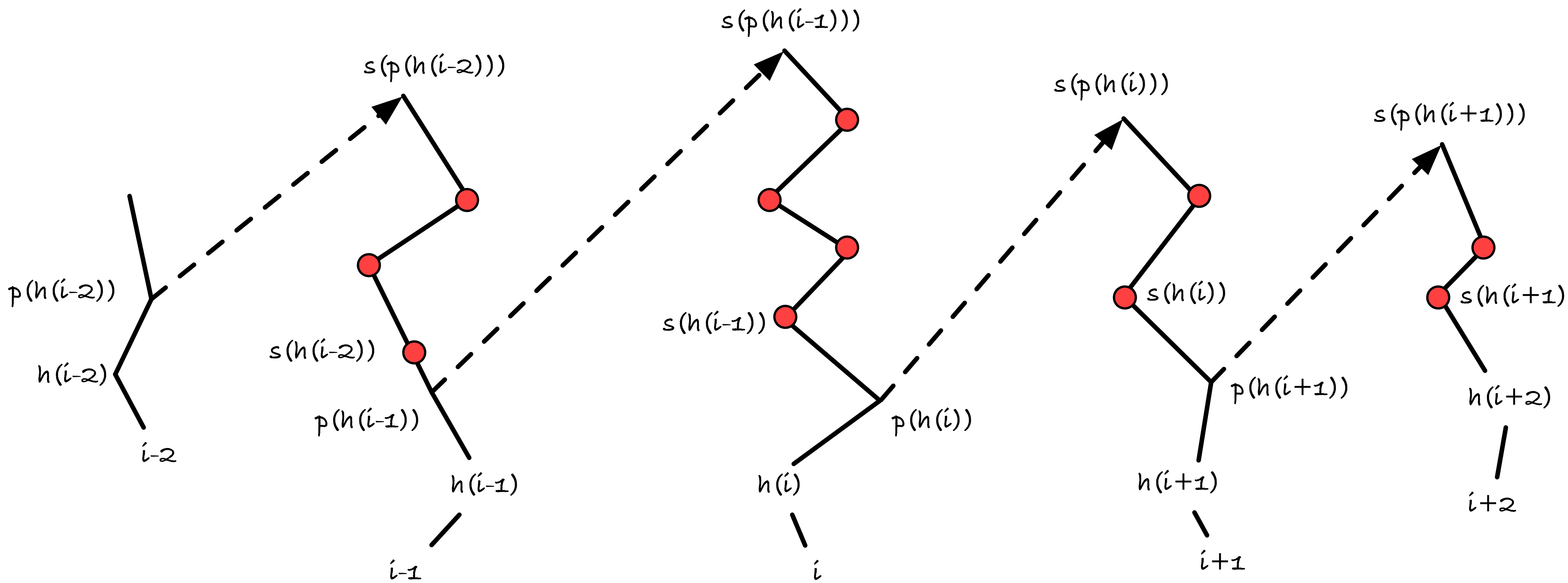


Jumps add up to O(n)

Scans add up to O(n)

What about fast scan?

Fast scan time usage



It’s hard to add up all the fast scans (i.e. I don’t know how to do it).

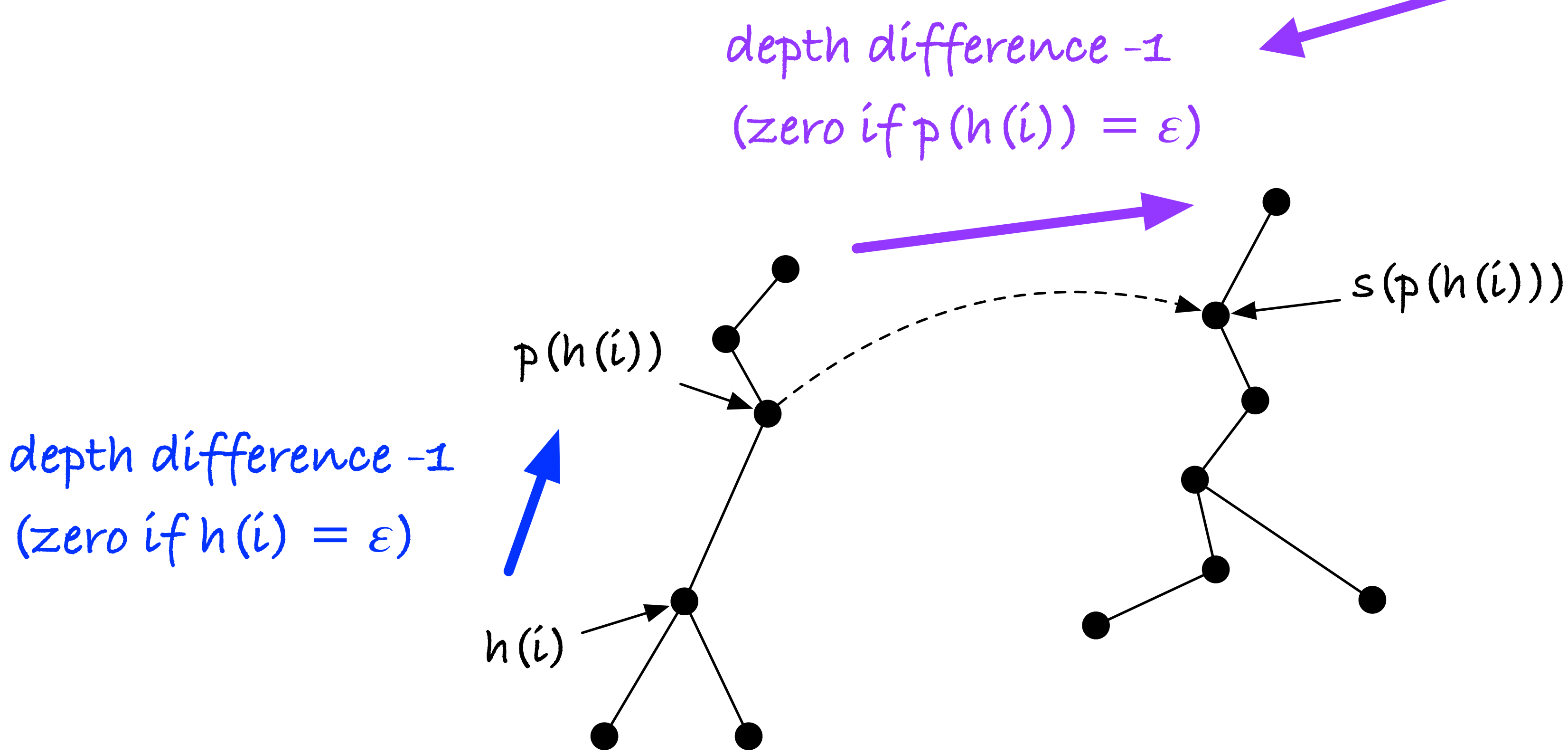
But we can put a bound on how deep you can go by putting a bound on how much you move up towards the root…

Fast scan time usage

- The max depth you can reach is n+1 (that's the longest string)
- If you want to move downwards more than that, you must first move upwards towards the root again.
- How much do we move upwards?

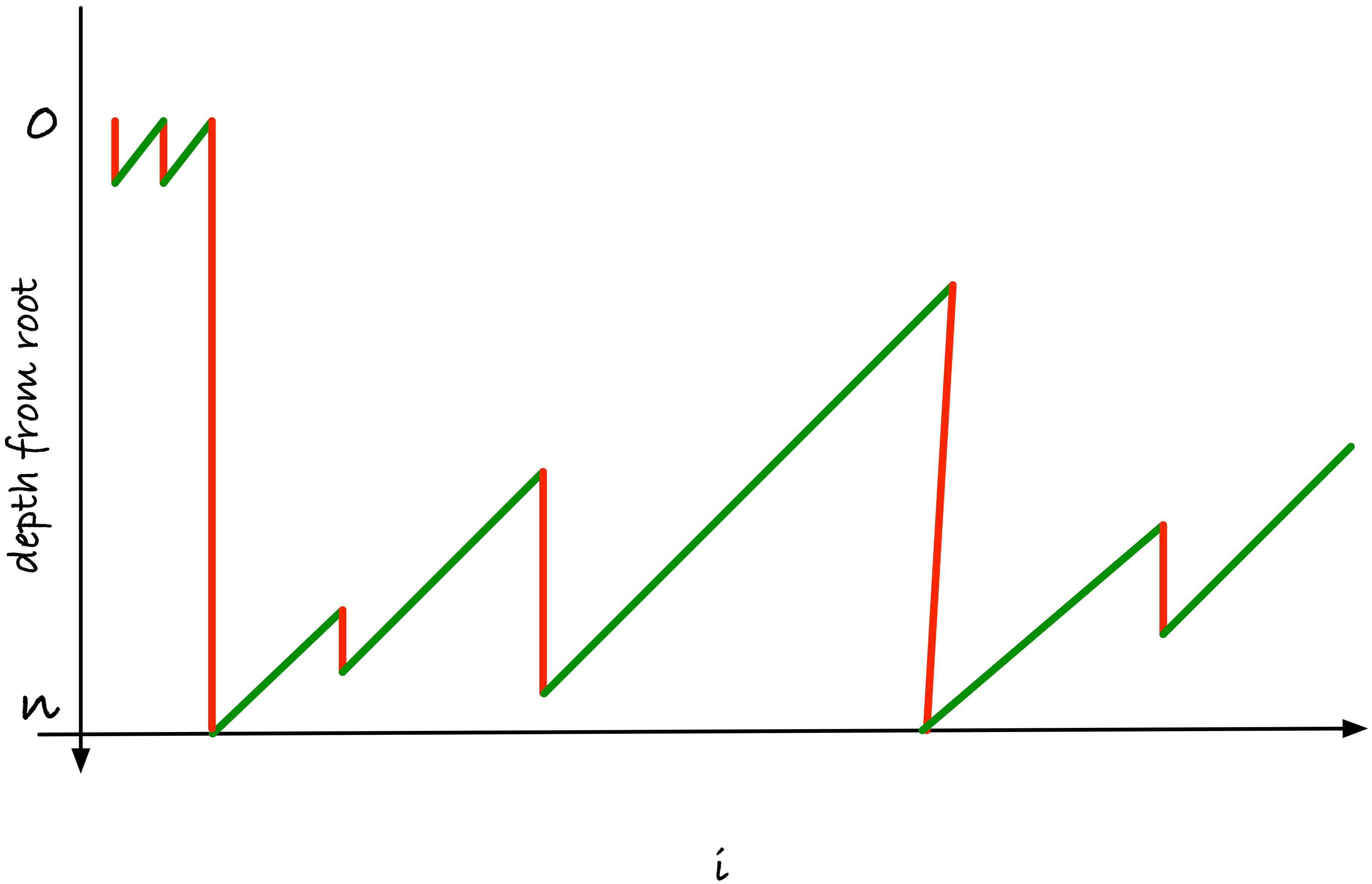
Fast scan time usage

Exercise



Max depth decrease per iteration: -2

Fast scan time usage



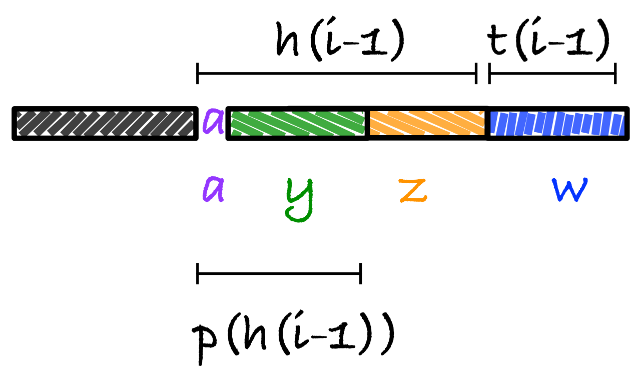
Moving along pointers
(max -2 per iteration)

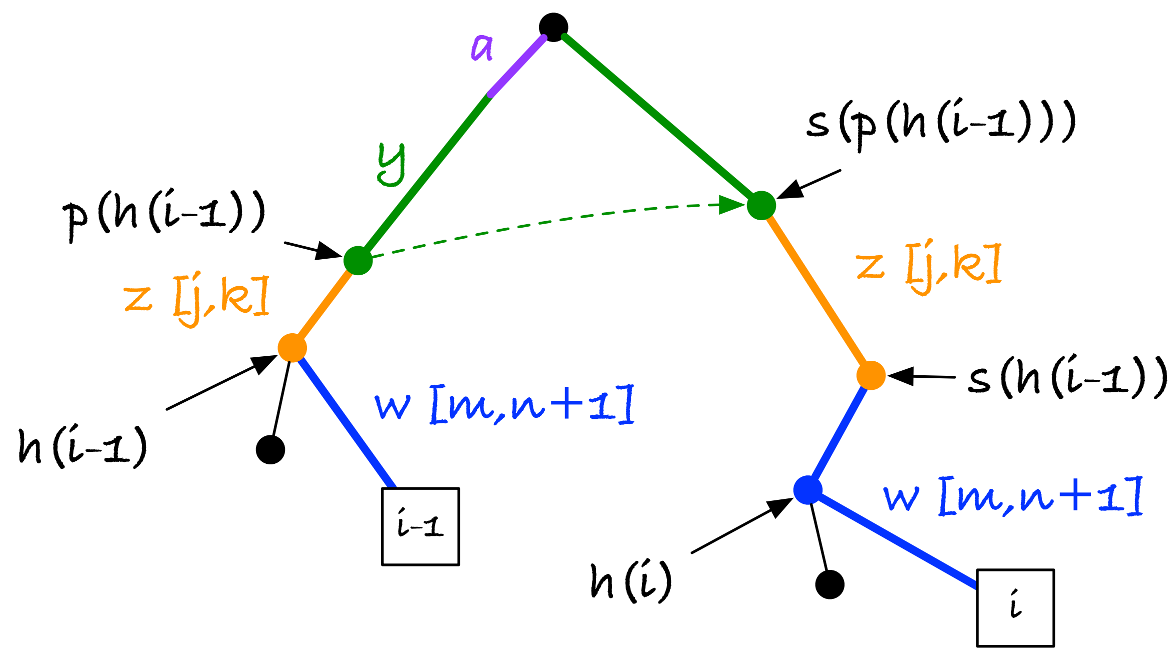
Cost of fast scan
$\leq n + 2n \leq O(n)$

Getting the strings

(and special cases)

General case

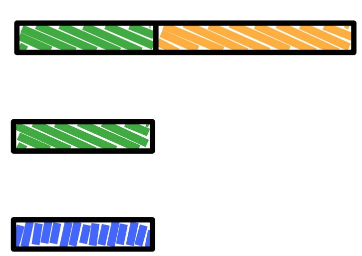




s(h(i-1)) = yz

s(p(h(i-1))) = y

t(i-1) = w



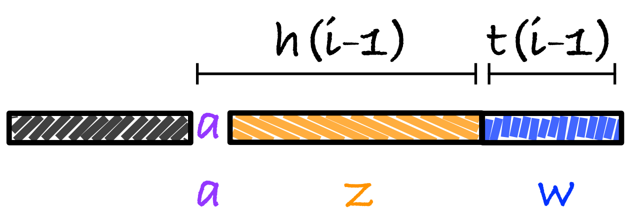
z = x[j:k]

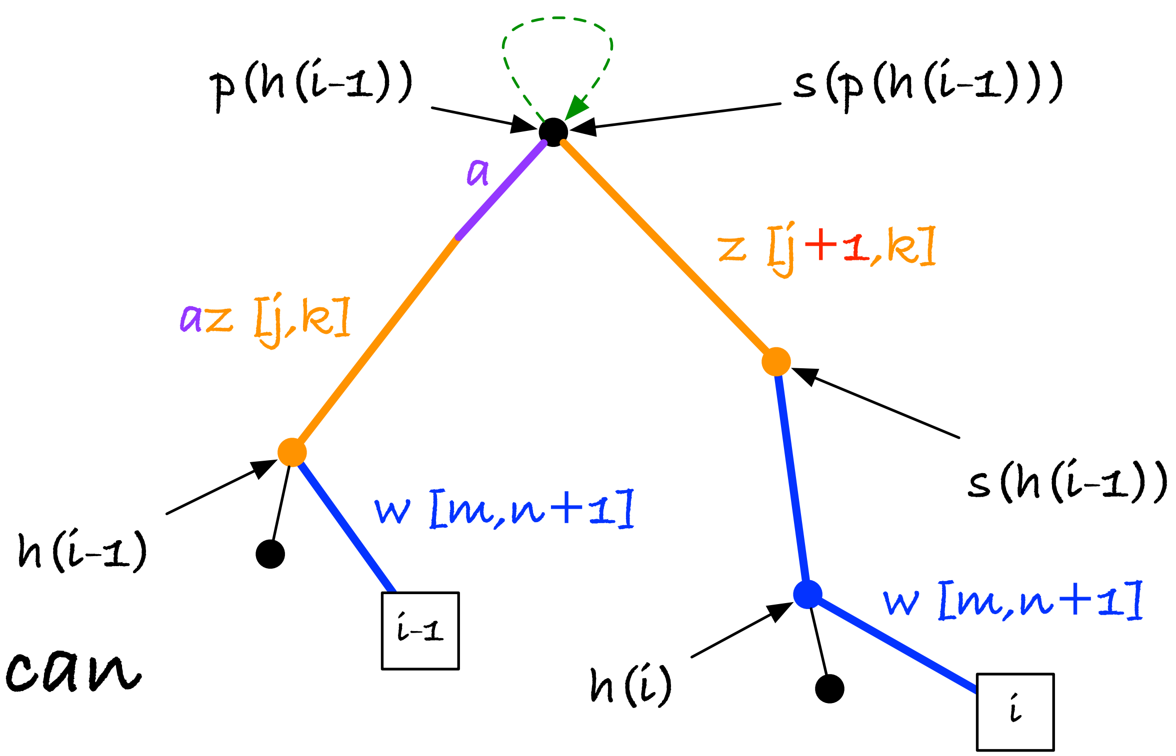
w = x[m:]

Getting the strings

(and special cases)

Parent is root



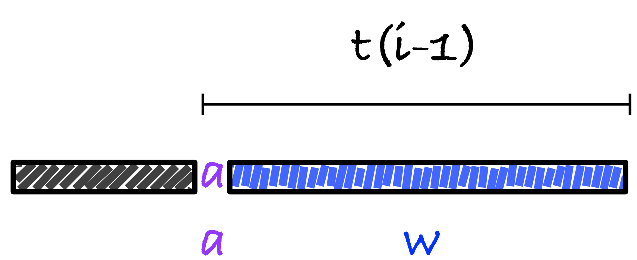


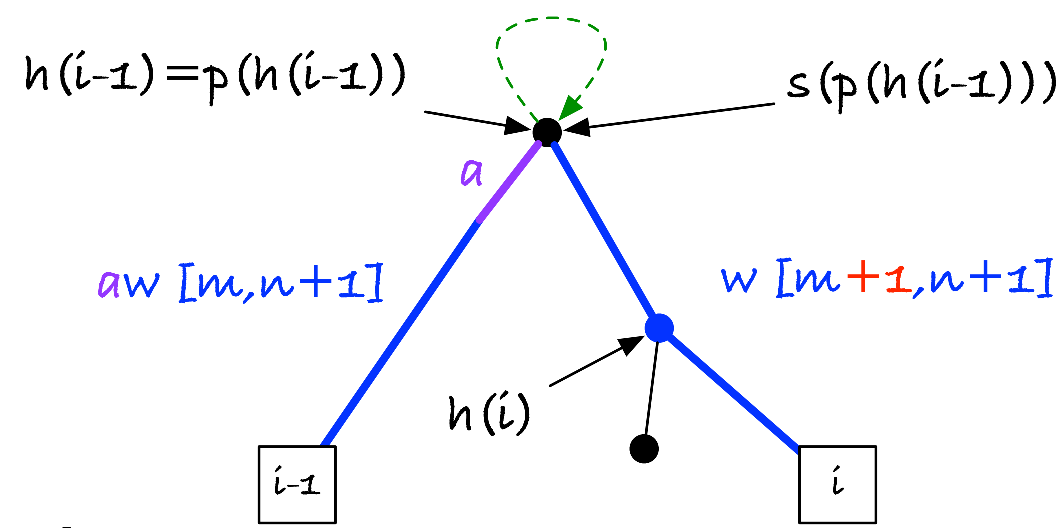
If you start your fast scan in the root, start at +1

Getting the strings

(and special cases)

Head is root





If you start your slow scan in the root, start at +1

Summary

- Iteratively build compacted tries (like the naïve algorithm)
- Use suffix links to jump past some of the search [O(1) per iteration]
- Use fast scan for initial part of search [amortised O(n) total]
- Slow scan the rest [non-overlapping strings, total O(n)]
- Now you can build suffix trees in linear time (and learned some new tricks you can use again later)

That's all Folks!