Lec13-SkipLists

Tuesday, September 26, 2023 10:41 AM

We've now spent a lot of time on graphs, which are of course very important, and are useful in illustrating a variety of algorithmic techniques.

Let's now apply some of those ideas to other structures.

List: [5, 7, apple, ☆, 100, ...]

What ops do we want to support?

L = [5, 7, ~~[524]~~, 999, 100, 200, 234]

get(L, i) or L[i] — L[3] = 999

insert(L, k, i) — insert(L, 200, 3) — insert item k before item with index 3.

delete(L, i) — delete(L, 2)

length(L) = 7

split(L, i) — split(L, 3) = [5, 7, 200], [999], [100, 200, 234]

find(L, k) — find(L, 7) = 1 (find index with value k)

copy(L) - return a full copy of list

sort(L) - return a sorted copy of L

Just like with graphs, we can implement in several different ways depending on what operations we care about.

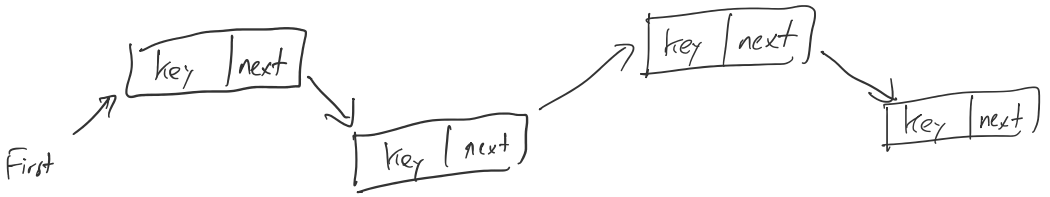
Ex. Use contiguous array in memory.

fast: get, length, split, copy

slow: insert, delete, search

important: copy takes the same amt of time as delete, but that's fine.

Ex. Linked list where records not adj. in memory, but use pointers instead.



good: insert, delete, don't need to preallocate memory, split, ...

bad: get, search, (jumping to middle is hard)

Optimizing search: Sorted array

[1, 5, 9, 15, 20, 50, 72, 99, 105, 200, 201]

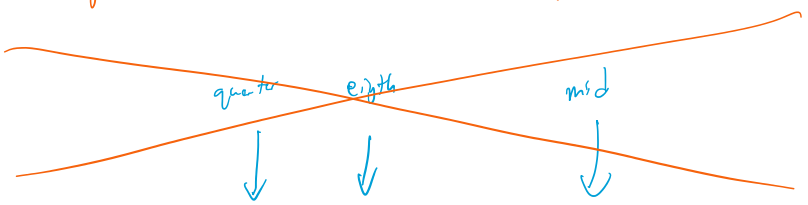
2 3 1

length = 11

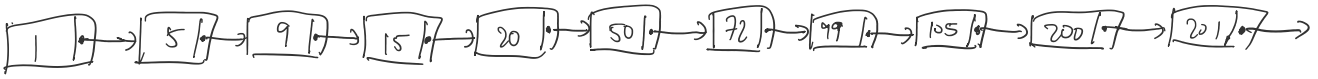
find index of key 15: start in middle, then go left or right using binary search

Can we do something similar with a sorted linked list?

Problem is we don't have pointers to the middle of list.

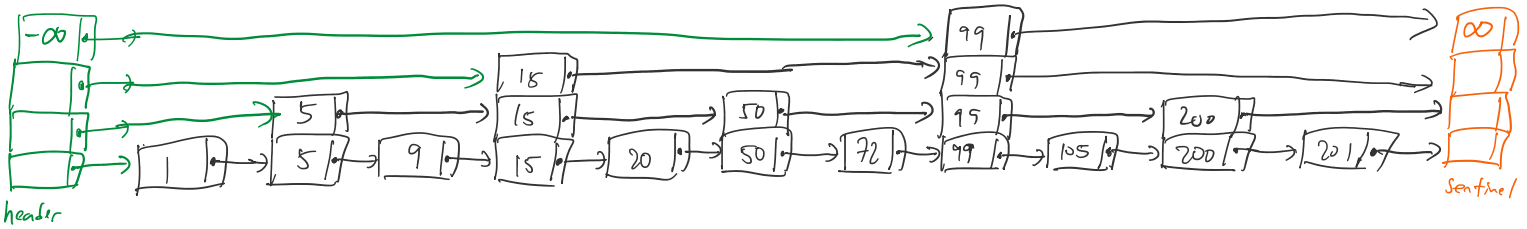


If we wanted to jump around, we would need pointers to every node



Perfect Skip List (generalization of sorted linked list)

- Promote half of nodes to higher level, repeatedly
- Keep headers & sentinels in every level
- Nodes contain between 1 & log(n) pointers
- Higher levels let you skip over nodes



Search: If k = key, done.
If $k <$ next key, go down.
If $k \geq$ next key, go right.

ex search 15:

top level start.
go down, since $15 < 99$
go right, since $15 \geq 15$
finish.

ex search 201:

top level, right to 99
go down a level since $201 < \infty$
go down a level, since $201 < \infty$
go right to 200 since $201 \geq 200$
go down a level, since $201 < \infty$
go right to 201, since $201 \geq 201$
finish.

We scan until we would skip the desired key, then go down.

$O(\log n)$ levels because each level is half as big.

Visit at most 2 nodes per level, because otherwise would've done it on a level higher.

$\Rightarrow$ $O(\log n)$ search time.

But how is this more useful than a sorted array?
insert & delete get broken because of the structure of heights

Notice: only runtime depends on the exact halving at each level.
Search would still work correctly if you insert or delete a node.
In worst case, just devolves back to a sorted linked list $O(n)$ search.

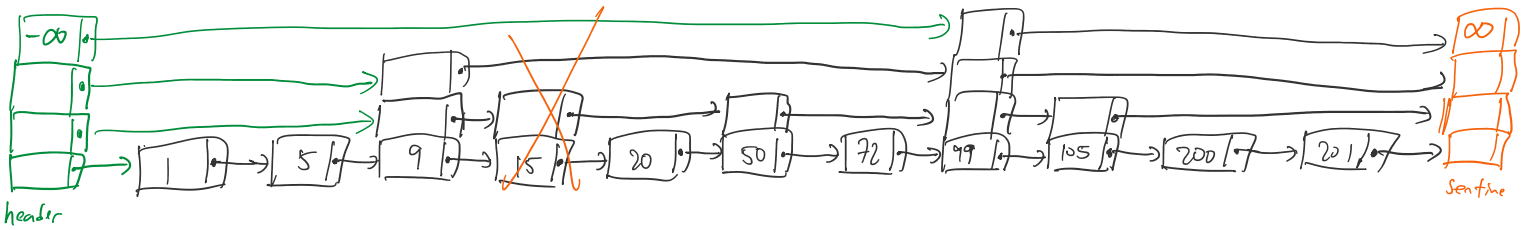
Idea: relax halving requirement to halving expectation

Skip Lists (1990, Pugh): randomized data structure w/ expected $O(\log n)$ search.

- Key: randomly promote nodes with $1/2$ prob, repeating until failure.

$\Rightarrow$ expected # nodes at level 0: n
level 1: $\frac{n}{2}$
level 2: $\frac{n}{4}$
$\vdots$

Also, expect that nodes are roughly evenly distributed.



deletion finds item as usual & removes it.

Insertion: find k
insert node in level 0
let $i = 1$

insert node in level v
let i = 1
while FLIP() = "heads":
    insert node into level i
    i++

} insertion into linked list right after last visited node in that level along finding path

Level structure is independent of keys inserted, so no bad sets of keys

Some miniscule probability we degenerate to linked lists.

Analysis: $\mathbb{E}[\#\text{levels}] = O(\log n)$

$\mathbb{E}[\#\text{ nodes level } 1] = n/2$

$\mathbb{E}[\#\text{ nodes level } 2] = n/4$

$\vdots$

$\mathbb{E}[\#\text{ nodes level } \log n] = 1$

What about number of steps we have to take at each level?

In search path, always enter node at its highest level.

So walking backward, always go up as much as possible

Prob can move up in reverse walk: 0.5

Always move up or left in each reverse step.

$\Rightarrow$ in expectation $2j$ steps to move up j levels

$\Rightarrow$ $2 \log n$ steps in expectation to get to top level.

Implementation notes: Nodes have variable size, but constant after creation.

Variable number of levels, but the max level is $O(\log n)$ w.h.p.