Lec15-skiplists

Thursday, September 26, 2024 3:37 PM

We saw that binary search trees can become unbalanced, which can lead to bad runtimes. Here we give a randomized BST-like search structure.

(ADT)

List: [5, 7, apple, ☆, 100, ...]

What ops do we want to support?

L = [5, 7, ~~15~~ 44, 999, 00, 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 key k) ← search operation

copy(L) - return full copy of list

sort(L) - return sorted copy of list

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

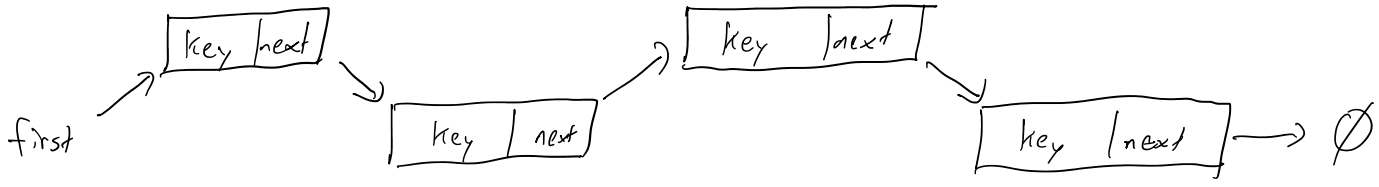
Ex. Use contiguous array in memory

fast: get, length, split, copy

slow: insert, delete, search

note: copy takes same amount of time as delete, so fast/slow is relative to time in alternative implementations

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



good: insert, delete, split (don't have to worry about mem allocation)

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

Optimizing search: Sorted array

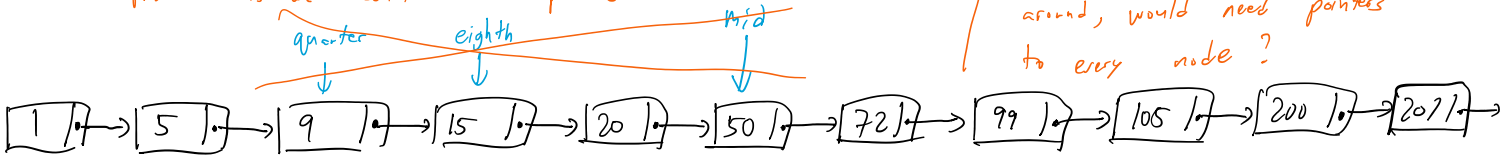
[1, 5, 9, 15, 20, 50, 72, 99, 105, 200, 201] length = 11

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

Can we do the same thing with sorted linked list?

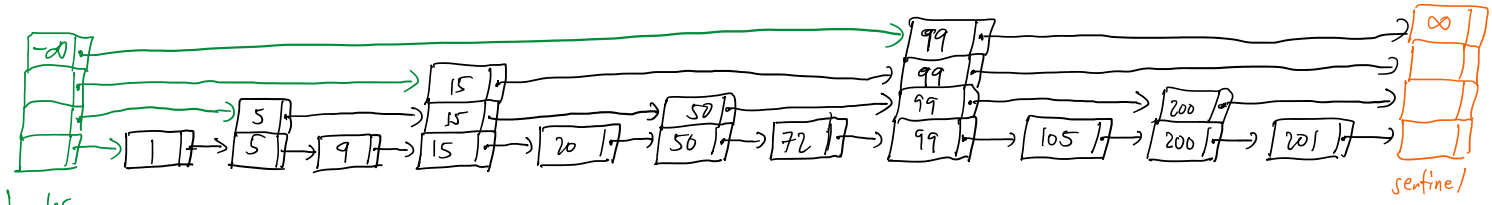
Problem is we don't have pointer to middle of list.

If we wanted to jump around, 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 each 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 ≥ next key, go right

ex. search 15:

top level start

go down, since 15 < 99

go right, since 15 ≥ 15

finish

ex. Search 201:

to level, right to 99

go down a leaf, since 201 < ∞

go down a leaf, since 201 < ∞

go right to 200, since 201 ≥ 200

go down a level, since 201 < ∞

go right to 201, since 201 ≥ 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 traveled on higher level.

=> 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 exact halving at each level.

Search would still work correctly if you insert or delete a node.

In worst case, just devolves down to a sorted linked list O(n) search.

Idea: relax halving requirement to halving expectation.

Skip Lists (1990, William Pugh):

randomized data structure with expected O(log n) search.

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

=> expected # nodes at level 0: n

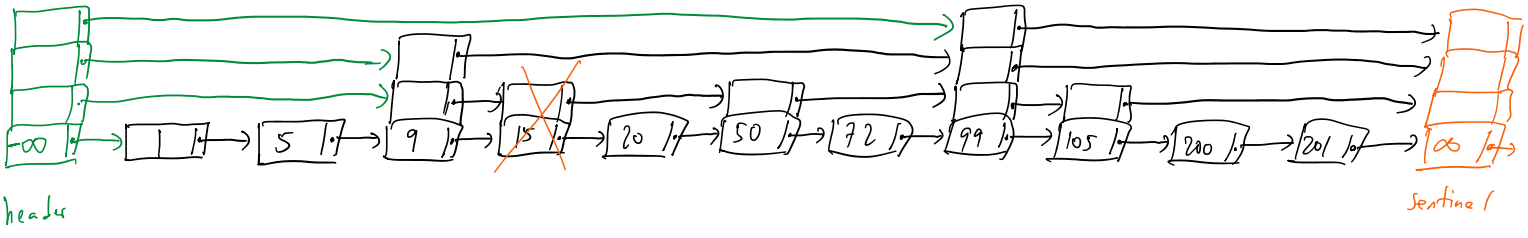
level 1: $\frac{1}{2}n$

level 2: $\frac{1}{4}n$

level 3: $\frac{1}{8}n$

⋮

Also, expect that nodes are roughly evenly distributed.



delete: find item & remove it

Insertion: find k
insert node in level 0
let i = 1
while FLIP() = "head":
insert node into level i

} Insertion into linked list

```
while FLIP() = "head":
    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 set of keys. Some miniscule probability we degenerate to linked lists

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

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

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

$\mathbb{E}[\#\text{ nodes level } 3] = \frac{n}{8}$

$\vdots$

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

What about how many steps we have to take at each level?

In search path, always enter node at its highest level

So walking backwards, 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.