CS280-Data Structures

Skip Lists

Overview

- Linked List pros and cons
- Skip Lists (High Performance Linked Lists)
- Structure of a Skip List
- Implementation Details
- Insertion
- Deletion
- References and further reading

Linked Lists

- Pros:
  - Efficient to insert/remove anywhere in the list
  - Easy to grow and shrink at runtime
  - Only use as much memory as we need

- Cons:
  - Allocating and de-allocating individual nodes
  - No random access to elements, locating a random element is $\boldsymbol{O(N)}$(even if sorted)

Simple analysis

- It's easy to solve the first con simply by using some kind of *object allocator*.
- The second con can also be "solved". (Not solved, per se, but minimized.)
- One way we've seen attempting to "solve" the second problem is with some king of balanced tree. This gives us worst case of $\boldsymbol{O}(\lg \boldsymbol{N})$, which is significantly better than $\boldsymbol{O}(\boldsymbol{N})$.
- Using trees comes at a cost in both runtime (re-balancing) and implementation (code complexity).
- We'd like our data structures to have the benefits of linked lists and trees, but with less runtime and implementation costs.
- The result will be some kind of *super* linked list or *high-performance* linked list

Skip Lists (High Performance Linked Lists)

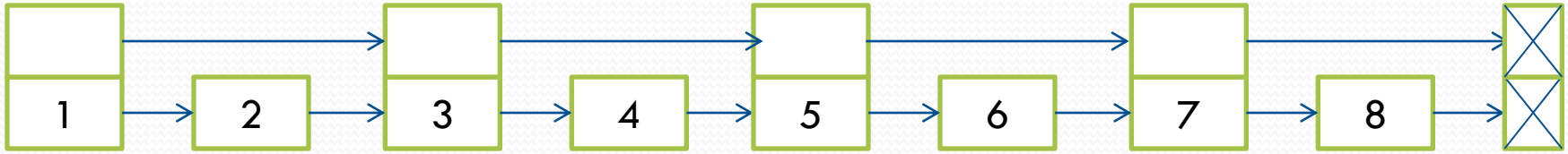
- The fundamental problem with linked lists is that during a search, we must look at every node in the list that comes before the target node, even if the list is sorted (No binary search methods are possible.)

Searching for "6" in the list will require to look at 5 nodes before reaching node "6". In other words, we have to look at "1", "2", "3", "4", "5".

- This is the part that skip lists are trying to improve.

Skip Lists (High Performance Linked Lists)

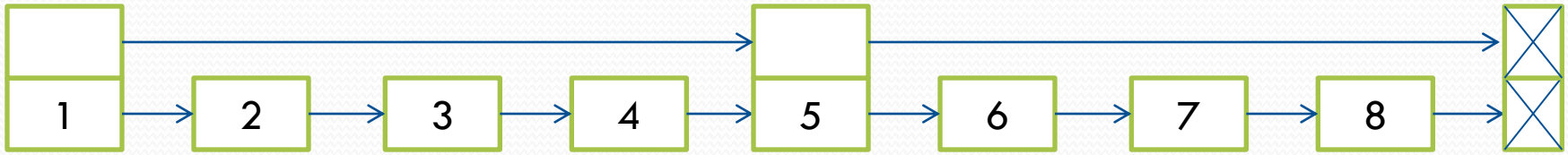
- If we create some nodes with 2 pointers, we can have them point "further" into the list, say 2 nodes ahead, this lets us move 2 nodes at a time, effectively



In this case, we are jumping two nodes at a time, looking at "1", "3", "5", "7", then backing up to "6".

Skip Lists (High Performance Linked Lists)

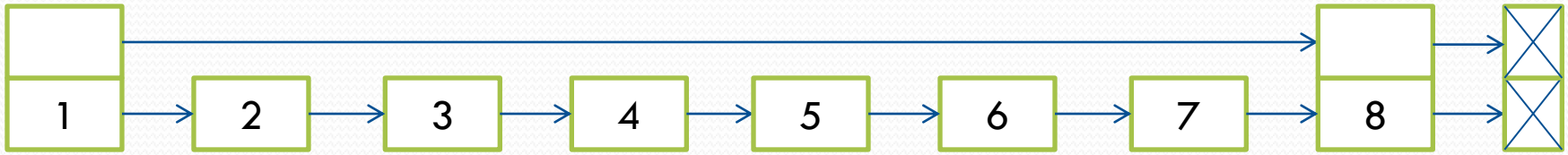
- We can take this notion further and move ahead 4 nodes at a time instead.



In this case, we are jumping 4 nodes at a time, looking at "1, "5", then backing up to "6".

Skip Lists (High Performance Linked Lists)

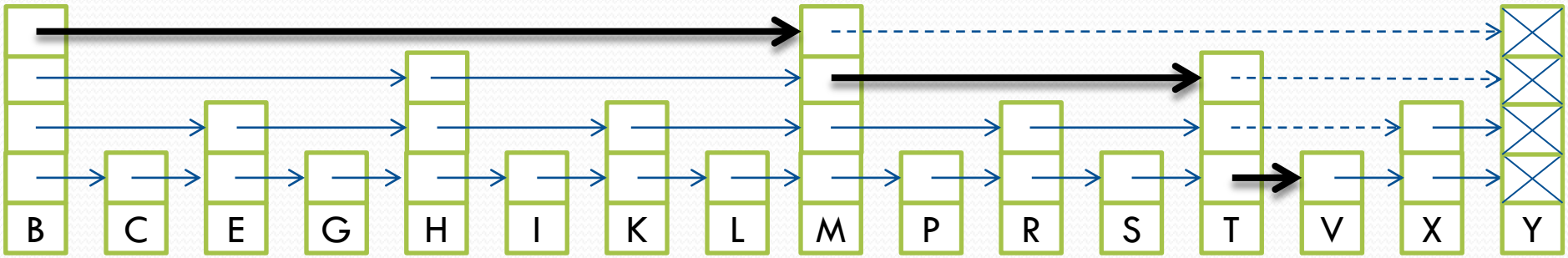
- How about 8 nodes?



In this case, we are jumping 4 nodes at a time, looking at "1, "5", then backing up to "6".

Skip Lists (High Performance Linked Lists)

- How about All combined?



- The **bolded** path shows how we would traverse the list in order to find "V" in the list. We would visit:

M, Y, T, Y, X, V

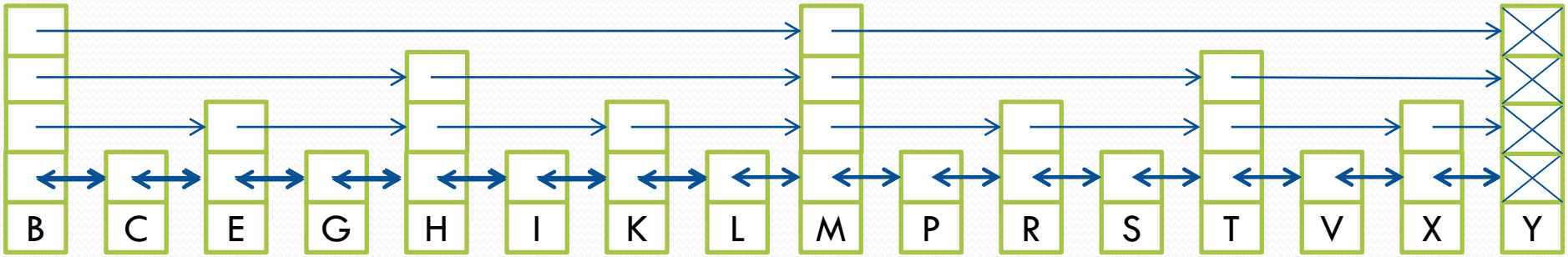
- By having multiple *next pointers*, we can vary the speed at which we move through the list. Kind of a built-in *fa*

Structure of a Skip List

- There are several key factors involved in building a skip list:
  - Should the list be single or double linked?
  - How do we choose the size of our skip intervals?
  - How do we determine which nodes will be of what type>
  - How do we deal with different node sizes which implies different types? (A pointer can only point to one type of node)

Structure of a Skip List

- At the base level *(level 0)*, we have a double-linked list where nodes contain a *next* and *previous* pointer:
  - At level 1, we have forward pointers that jump ***two*** nodes at a time.
  - At Level 2, forward pointers jump ***four*** nodes at a time.
  - At level 3, forward pointers jump ***eight*** nodes at a time.



Structure of a Skip List

- Given the scheme above:
  - Each level skips twice as many nodes as the previous level.
  - Level 0 moves 1 node at a time, level 1 moves 2 nodes, level 2 moves 4, level3 moves 8 etc…
  - We can observe that at a given level, n, we can move 2^n nodes at a time.
  - It's not hard to see that the complexity of finding an item in the worst case is $\boldsymbol{O(\log_2 N)}$.

Structure of a Skip List

- How many levels do we need?
  - Computer memory is finite, so there is a practical limit to how many levels we can have in our nodes.
    - At level 32, we'd be moving over 4 billion nodes at a time, so clearly, we have a limit to how large the nodes can grow.
  - Pointers required:
    - A level 0 node needs to store 2 pointers. (next and previous).
    - Level 1 nodes need to store 3 pointers. (next, previous, 1 forward).
    - At level 2, we store 4 pointers. (next, previous, 2 forward)
    - At level n, we need to store $n + 2$ pointers. (next, prev, n forward)

Skip factors

- If we count the number of links:
  - Assuming there are N links at level 0 ($2N$ if double-linked), $\frac{N}{2}$ links at level 1, $\frac{N}{4}$ links at level 2, $\frac{N}{8}$ links at level 3 and $\frac{N}{2^L}$ links at level L, which gives us:

$$N + \frac{N}{2} + \frac{N}{4} + \ \ldots + \frac{N}{2^L} = N\left(1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots\right) = 2N$$

  - In this example, we've been using a *skip factor* of 2, meaning that each successive level skips over 2 times as many nodes as the previous level. We can set our skip factor to any number, such as 3 or 4

Skip factors

- *Arbitrary skip factors:*

$$\frac{N}{t^0}+\frac{N}{t^1}+\frac{N}{t^2}+\cdots=N\left(1+\frac{1}{t^1}+\frac{1}{t^2}+\frac{1}{t^3}\right)=\frac{t}{t-1}*N$$

t	Pointers
2	$2N$
3	$\frac{3N}{2}$
4	$\frac{4N}{3}$
5	$\frac{5N}{4}$
6	$\frac{6N}{5}$

Skip factors

- With an arbitrary skip factor, we can compute the number of nodes at a certain level:

$$\frac{(SF - 1)N}{SF^{L+1}}$$

Skip Factor 2

Level	Nodes	Level	Nodes
0	131072	9	256
1	65536	10	128
2	32768	11	64
3	16384	12	32
4	8192	13	16
5	4096	14	8
6	2048	15	4
7	1024	16	2
8	512	17	1

Skip Factor 4

Level	Nodes	Level	Nodes
0	196608	9	1
1	49152		
2	12288		
3	3072		
4	768		
5	192		
6	48		
7	12		
8	3		

Implementation Details

- For these examples, we'll assume that the data is simply a 4-byte void pointer:

```
const unsigned MaxSkipLevels = 32; // can skip 4 billion nodes!
typedef struct SLNode* SLNodeArray[MaxSkipLevels];

struct SLNode
{
  void *data;          // The arbitrary data in the node
  unsigned level;      // The type (level) of node
  SLNode *prev;        // Previous pointer (assume doubly-linked)
  SLNodeArray next;    // Array of "forward" pointers
};
```

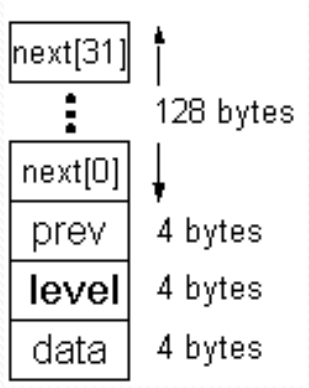
Note: Although the structure above appears to have a lot of wasted space, we won't ever actually create an SLNode directly. We will allocate "raw" memory and treat it as an SLNode

Implementation Details

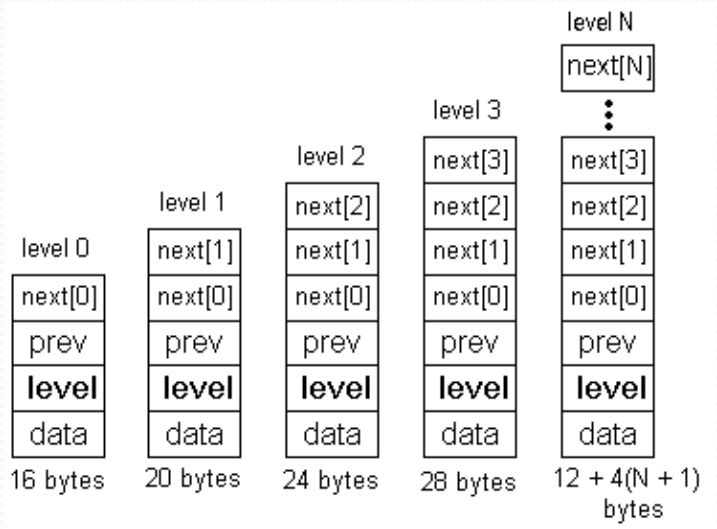
- Note that this is very inefficient (memory-wise):

```
node = new SLNode // allocates 140 bytes for all nodes in the list
```

- Since most (94%) of the nodes are less than level 4 this wastes far too much memory



SLNode created by declaration



SLNode representing different levels

Implementation Details

- Memory-efficient code to allow variable-sized SLNode structures:

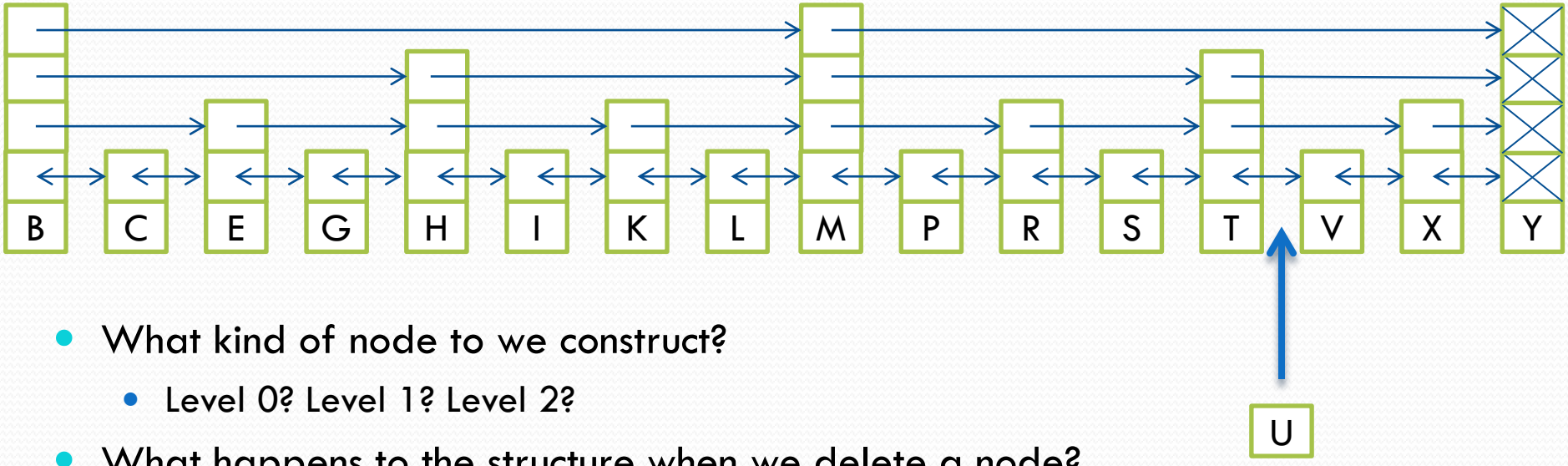
```
SkipList::SLNode* SkipList::allocate_node(unsigned level) const throw(SLException)
{
    // sizeof(data + level + prev) = 12
  unsigned size = 12 + (level + 1) * sizeof(void *);
  SLNode *node = 0;
  try
  {
    node = reinterpret_cast<SLNode *>( new char[size] );
  }
  catch (const std::bad_alloc &)
  {
    throw SLException(SLException::E_NO_MEMORY, "allocate_node: Out of memory");
  }

    // initialize fields
  node->level = level;
  node->prev = 0;
  for (unsigned int i = 0; i <= level; i++)
    node->next[i] = 0;

  return node;
}
```

Inserting into a Skip List

- Consider this example, suppose we want to insert **U** into this list:



- What kind of node to we construct?
  - Level 0? Level 1? Level 2?
- What happens to the structure when we delete a node?
- Since we are inserting/deleting random data, how can we be sure that we can build the proper list with the exact kinds of nodes in exact places?

Maintaining the Skip Lists

- How can we build the proper list with the exact kinds of nodes in exact places?
  - We can't *(not without a lot of work)*.
  - We can approximate. No need to be exact.
  - The exception is the node with the highest level which will have an extra count.

Level	Node Count	Frequency	Nodes
0	8	0.5	BEHKMRTX
1	4	0.25	CIPV
2	2	0.125	GS
3	1	0.0625	LY

A Perfect List with 2^{14} nodes

- This list is in practice, impossible (at least not without a lot of work).
- In practice the largest node having one extra count (for the tail node).
- What kind of node *do* we create when inserting an item?

Level	Node Count	Frequency
0	8192	0.5
1	4096	0.25
2	2048	0.125
3	1024	0.0625
4	512	0.03125
5	256	0.015625
6	128	0.0078125
7	64	0.00390625
8	32	0.001953125
9	16	0.0009765625
10	8	0.00048828125
11	4	0.00244140625
12	2	0.000122073125
13	1	0.0006103515625

Roll the dice!

- When we create an item, we simply roll the dice and see what kind of node (*level*) we get.
- We need our "dice" to have the frequencies(probabilities) shown above.
- So, in the example above, we'd want to create a level 0 node 50% of the time, a level 1 node 25% of the time, a level 2 node 12.5% of the time, etc…

Roll the dice!

```
unsigned place_level()
{
     // calculate the level for the new node
     unsigned Level = 0;
     double d = drand();// roll the dice (returns number 0 < x < 1)

     // maxLevel_ is the largest node currently constructed
     while( (Level <= maxLevel_) && (d < (1.0/baseFactor_)))
     {
          Level++;
          d = drand();
     }

     // if we've gone beyond the maximum level raise the max
     if(Level > maxLevel_)
          maxLevel_++;

     //return level
     return Level;
}
```

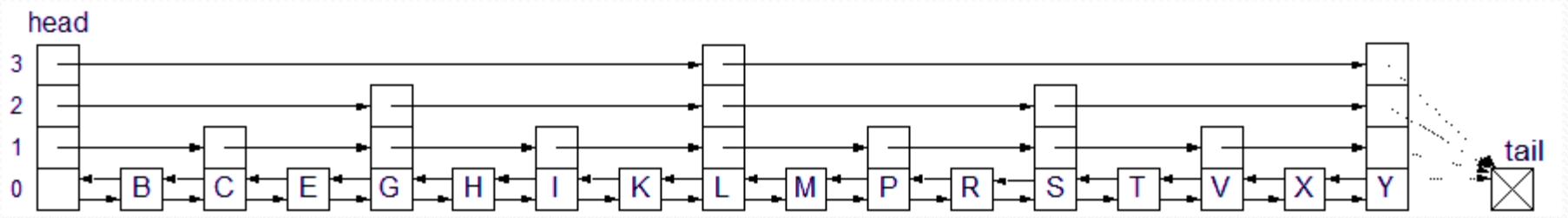
Sample run on 256K items

Statistics with base factor: 2

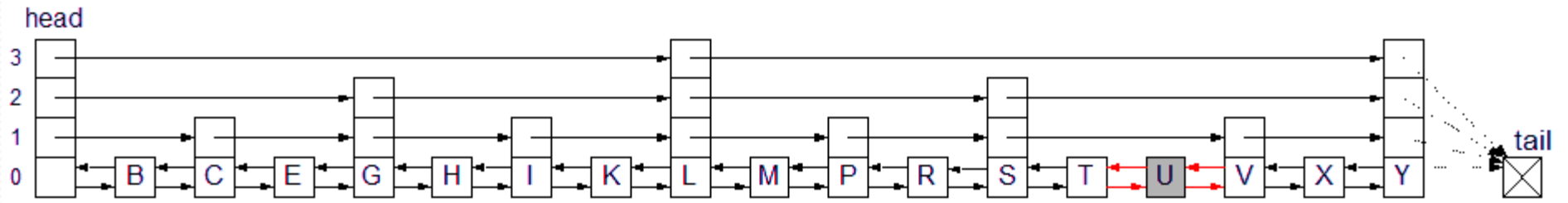
	Ideal	Actual	Ideal Probability	Act. Probability
Level 0 nodes:	131072	133590	50%	50.960540771484375%
Level 1 nodes:	65536	65926	25%	25.148773193359375%
Level 2 nodes:	32768	31846	12.5%	12.148284912109375%
Level 3 nodes:	16384	15630	6.25%	5.962371826171875%
Level 4 nodes:	8192	7652	3.125%	2.91900634765625%
Level 5 nodes:	4096	3859	1.5625%	1.4720916748046875%
Level 6 nodes:	2048	1816	0.78125%	0.69274902343749989%
Level 7 nodes:	1024	958	0.390625%	0.36544799804687506%
Level 8 nodes:	512	426	0.1953125%	0.162506103515625%
Level 9 nodes:	256	223	0.09765625%	0.0850677490234375%
Level 10 nodes:	128	106	0.048828125%	0.040435791015625%
Level 11 nodes:	64	61	0.0244140625%	0.0232696533203125%
Level 12 nodes:	32	24	0.01220703125%	0.0091552734375%
Level 13 nodes:	16	13	0.006103515625%	0.004959106445312 5%
Level 14 nodes:	8	10	0.0030517578125%	0.003814697265625%
Level 15 nodes:	4	3	0.00152587890625%	0.001144409179687 5%
Level 16 nodes:	2	1	0.000762939453125%	0.000381469726562 5%

Total nodes: 262144
Total pointers: 513961
pointers/nodes: 1.9606

Insert new node

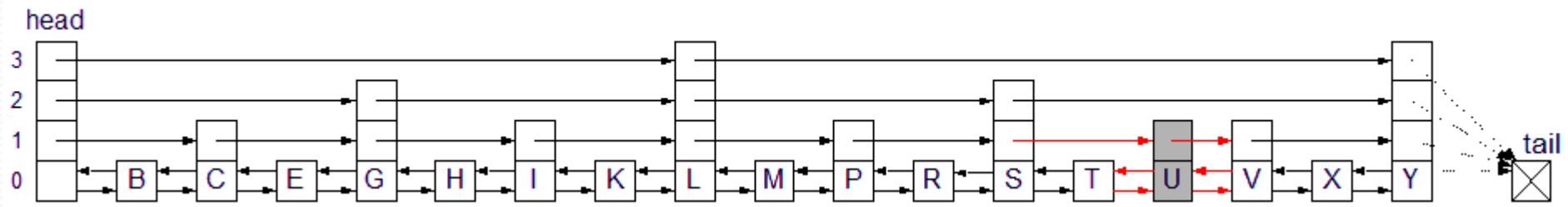


Original List

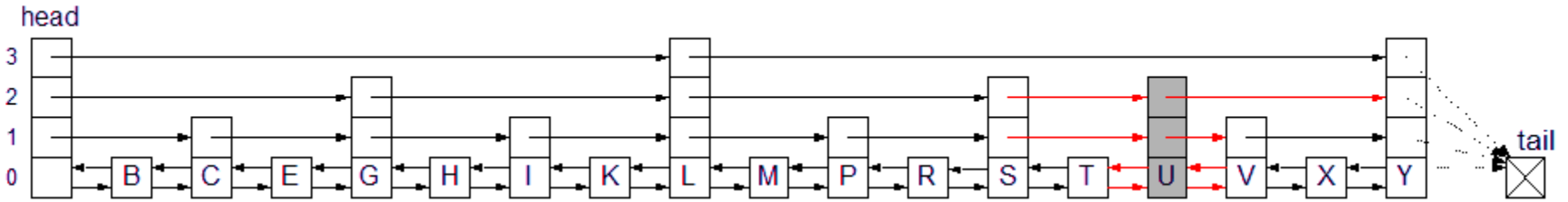


As a level 0 node

Insert new node

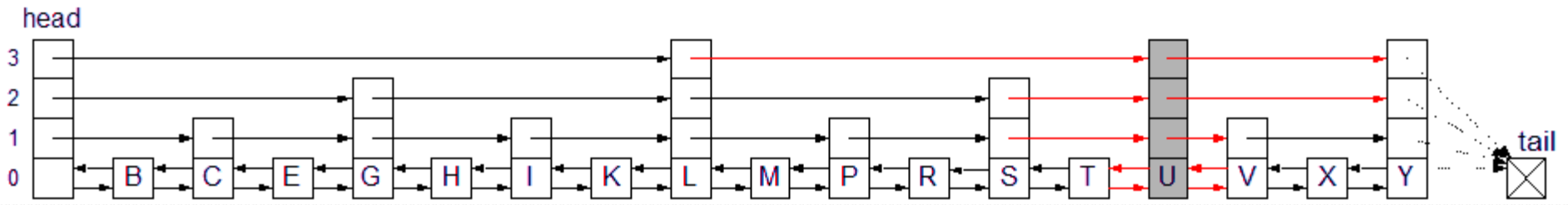


As level 1 node

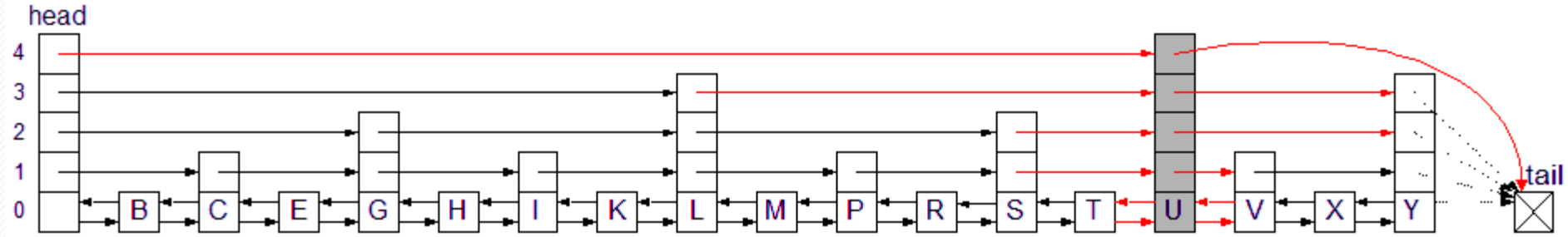


As a level 2 node

Insert new node



As level 3 node

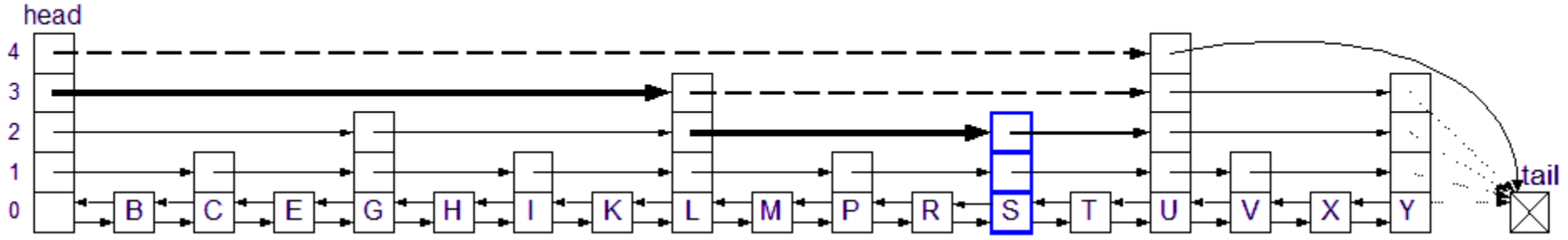


As a level 4 node

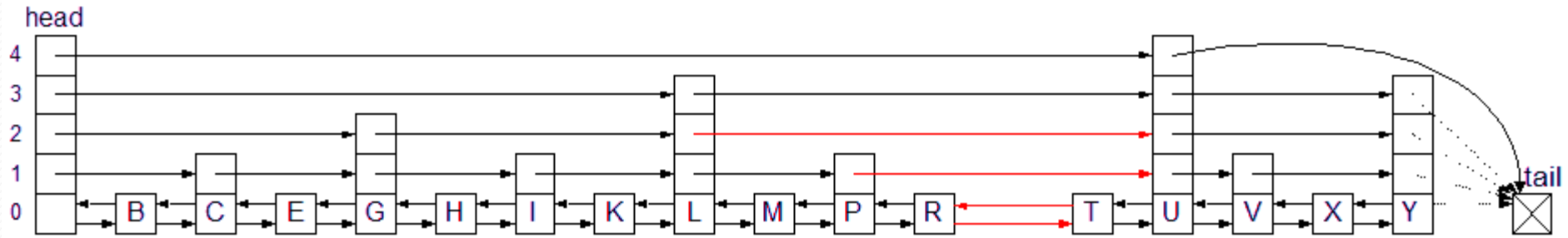
Notes:

- When inserting in a Skip list, we only need to keep track of all the nodes that led to the new one.
  - Update the upper level pointers depending on the level of the newly created node.

Deleting a Node:



Original List

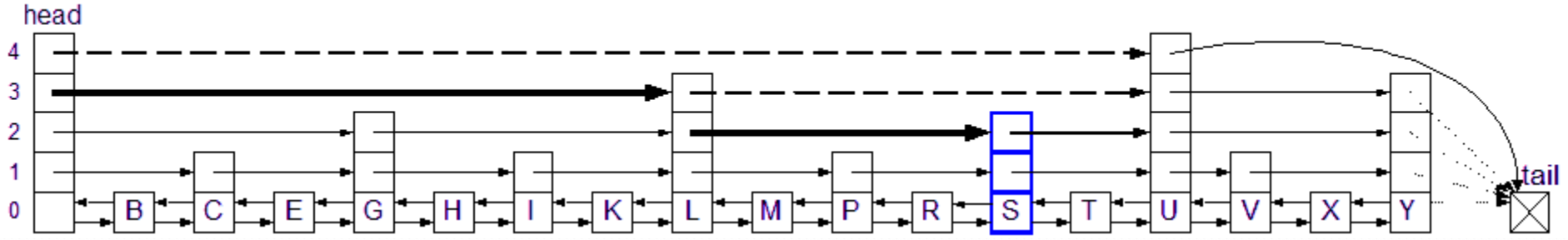


After deleting S

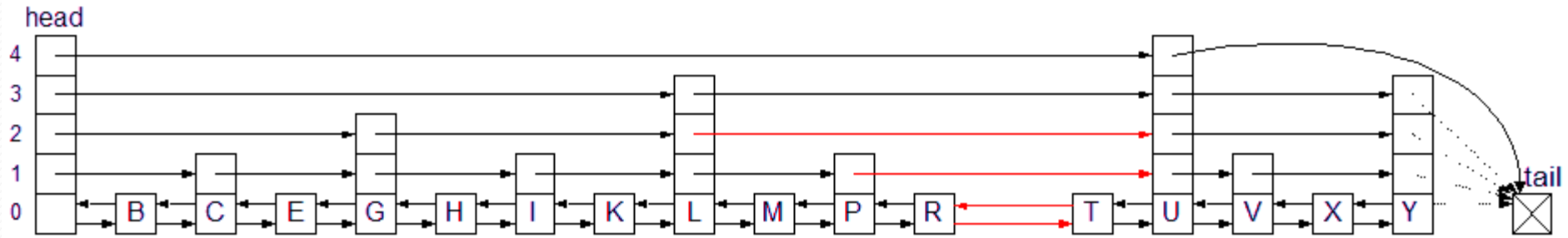
Deleting a Node

- Like insertion, we need to keep track of *all* the nodes that led to the new one so that we can update the pointers correctly.
- There is only slightly more work to be done due to the fact that not all of the nodes that lead to the deleted node were traversed during the din. In the example below, nodes **P** and **R** need to be updated, yet they didn't lead to **S**.

Deleting a Node:



Original List



After deleting S

Consideration

- Skip lists are another form of randomized algorithms.
  - Much like hash tables, we add randomness to help the performance.
  - We also trade memory for speed (More pointers per node == faster search).
- Compared to some balanced tree algorithms, the skip lists is much simpler.
- Most of the implementation complexity is in keeping track of the forward pointers when inserting/deleting.
- Object Allocator might be difficult to use with our *efficient* allocation scheme.
- What happens to our *locality of reference* that the object allocator provides?