Skip lists (Λίστες παράλειψης)

Skip lists

- We will now consider an approach to developing a fast implementation of symbol-table operations that seems at first to be completely different from the tree-based methods that we have been considering, but actually is closely related to them.
- The approach is based on a **randomized data structure (τυχαιοκρατική ή πιθανοτική δομή δεδομένων)** and is almost certain to provide **near-optimal performance** for the basic operations of the symbol table ADT.
- The data structure is called a **skip list (λίστα παράλειψης)**. It uses extra links in the nodes of a linked list to skip through large portions of a list at a time during search.

Example

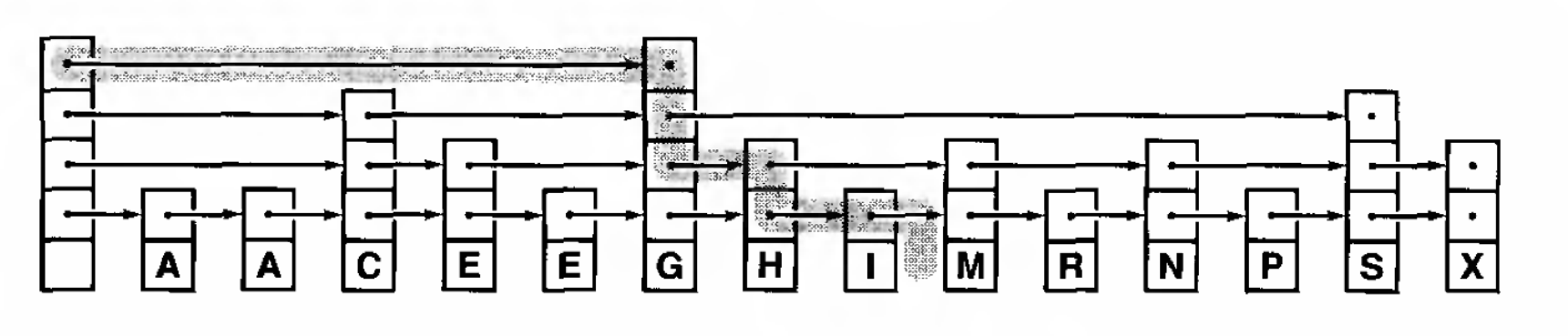
Notes

- The previous slide shows an example of a skip list where every third node in an ordered linked list contains an extra link that allow us to skip three nodes in the list.
- We can use the extra links to speed up search: we scan through the top list until we find the key or a node with a smaller key with a link to a node with a larger key, then use the links at the bottom to check the two intervening nodes.
- This method speeds up search by a factor of *3*, because we examine only *k/3* nodes in a successful search for the *k*-th node on the list.

Notes (cont'd)

- We can iterate this construction, and provide a second extra link to be able to scan faster through nodes with extra links, and so forth.
- Also, we can generalize the construction by skipping a variable number of nodes with each link. See the example on the next slide.

Example



Definition

- A **skip list** is an ordered linked list where each node contains a variable number of links, with the *i*-th links in the nodes implementing singly linked lists that skip the nodes with fewer than *i* links.

Skip list definition

```
typedef struct STnode* link;

struct STnode
  { Item item; link* next; int sz; };

static link head, z;
static int N, lgN;
```

Notes

- The element `next` of a skip list node is an **array of links**.
- The element `sz` of a skip list node is the **number of links** in the node.
- The variable `N` keeps the **number of items in the list.**
- The variable `lgN` is the **current maximum number of levels in a node of the skip list.**
- `z` is a **sentinel node** (we will see below how it is used).

Skip list initialization

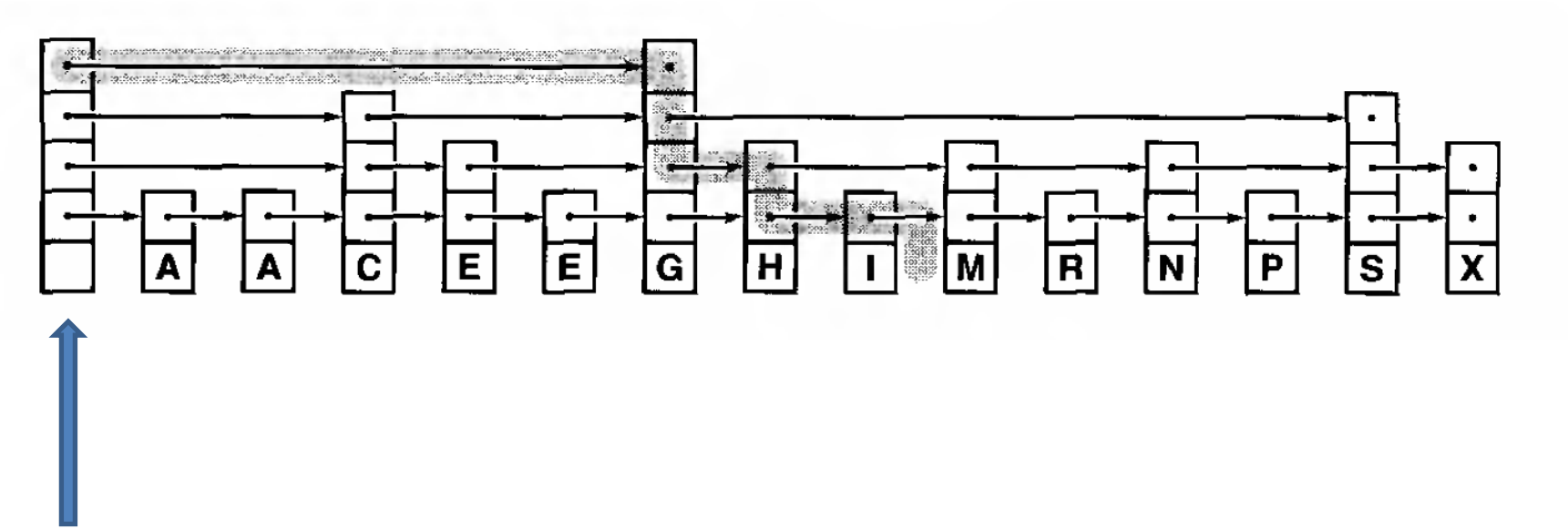
```
link NEW(Item item, int k)
  { int i; link x = malloc(sizeof *x);
    x->next = malloc(k*sizeof(link));
    x->item = item; x->sz = k;
    for (i = 0; i < k; i++) x->next[i] = z;
    return x;
  }

void STinit(int max)
  {  N = 0; lgN = 0;
     z = NEW(NULLitem, 0);
     head = NEW(NULLitem, lgNmax);
  }
```

Notes

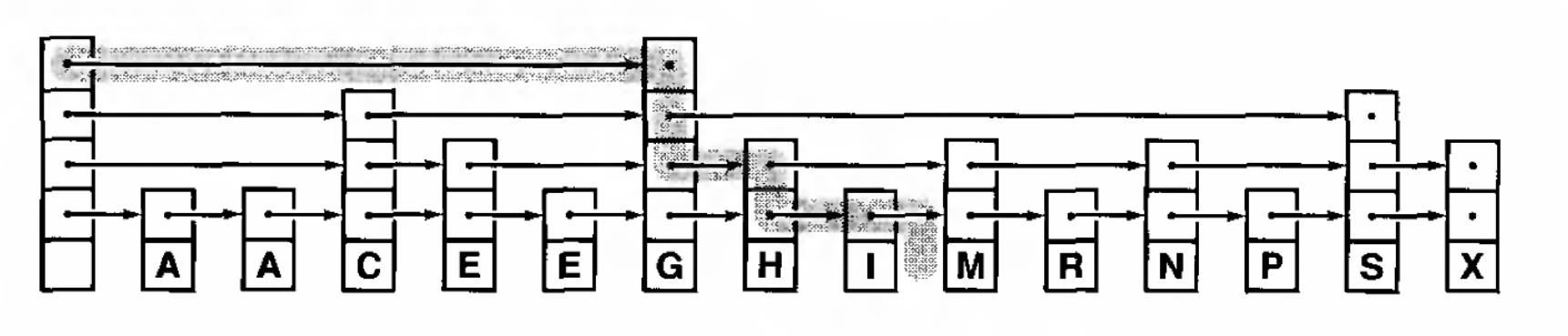
- Nodes in skip lists have an array of links, so `NEW` needs to allocate the array and to set the links to the sentinel `z`.
- The constant `lgNmax` is the **maximum number of levels that we allow in the list.** It might be set to `5` for tiny lists, or to `30` for huge lists.
- An **empty skip list** is a header node with `lgNmax` links, all set to `z`, with `N` and `lgN` set to `0`.
- In other words, to initialize a skip list, we build a header node with the maximum number of levels that we will allow in the list, with pointers at all levels to the sentinel node `z` **.**
- The sentinel node `z` has item `NULLitem` with key `maxKey` which is larger than all keys in the list (see later how it is used in search).

Example



Header node

Example: search for key L



Searching in skip lists

- To search in a skip list for a given key, we scan through the top list until we find the search key or a node that has a link to a node with a larger key. Then, we move to the second-from-the-top list and iterate the same procedure.
- If the next node has a key smaller than the search key the we continue our search in that node and iterate the procedure.
- We continue in this way until the search key is found or a search miss happens at the bottom level.

Searching in skip lists

```
Item searchR(link t, Key v, int k)
  { if (eq(v, key(t->item))) return t->item;
    if (less(v, key(t->next[k]->item)))
      {
        if (k == 0) return NULLitem;
        return searchR(t, v, k-1);
      }
    return searchR(t->next[k], v, k);
  }

Item STsearch(Key v)
  { return searchR(head, v, lgN); }
```

Notes

- For `k` equal to `0`, this code is equivalent to code for searching in singly linked lists.
- For general `k`, we move to the next node in the list on level `k` if its key is smaller than the search key and down to level `k-1` if its key is not smaller.
- To simplify the code, we assume that all the lists end with a sentinel node `z` that has item `NULLitem` with key `maxKey` which is larger than all keys in the list.

Notes (cont'd)

- The previous code is also **similar to binary search or searching in binary search trees:**
  - We test whether the current node has the search key.
  - Then, if it does not, we compare the key in the current node with the search key.
  - We do one recursive call if it is larger and a different recursive call if it is smaller.

Insertion in skip lists

- The first task that we face when we want to insert a new node into a skip list is to determine how many links we want that node to have.
- All the nodes have at least one link; we can skip t nodes at a time on the second level if one out of every t nodes has two links; iterating we come to the conclusion that we want one out of t^j nodes to have at least $j + 1$ links.

Insertion in skip lists (cont'd)

- To make nodes with this property, we **randomize**, using a function that returns i with probability $1/2^i$.
- Given i, we create a new node with i links and insert it into the skip list using the same recursive procedure as we did for search.
- After we have reached level i, we link in the new node each time that we move down a level.
- At that point, we have established that the item in the current node is less than the search key and links (on level i) to a node that is not less than the search key.

Insertion in skip lists (cont'd)

```
void insertR(link t, link x, int k)
  { Key v = key(x->item);
    if (less(v, key(t->next[k]->item)))
      {
         if (k < x->sz)
           { x->next[k] = t->next[k];
             t->next[k] = x;
           }
         if (k == 0) return;
         insertR(t, x, k-1); return;
      }
    insertR(t->next[k], x, k);
  }

void STinsert(Key v)
  { insertR(head, NEW(v, randX()), lgN); N++; }
```

Notes

- In the code of the previous slide, `insertR` is called with a second argument a node with a random number of links given by the function `randX`.
- The function `insertR` works similarly to `searchR`.
- When we reach the level `k=x->sz-1`, we link in the new node each time that we move down a level. This is done by the code inside the second `if` statement where `t` is linked with `x` which is linked with the node that used to come after `t`.

Randomization

```
int randX()
   { int i, j, t = rand();
     for (i = 1, j = 2; i < lgNmax; i++, j += j)
        if (t > RAND_MAX/j) break;
     if (i > lgN) lgN = i;
     return i;
   }
```

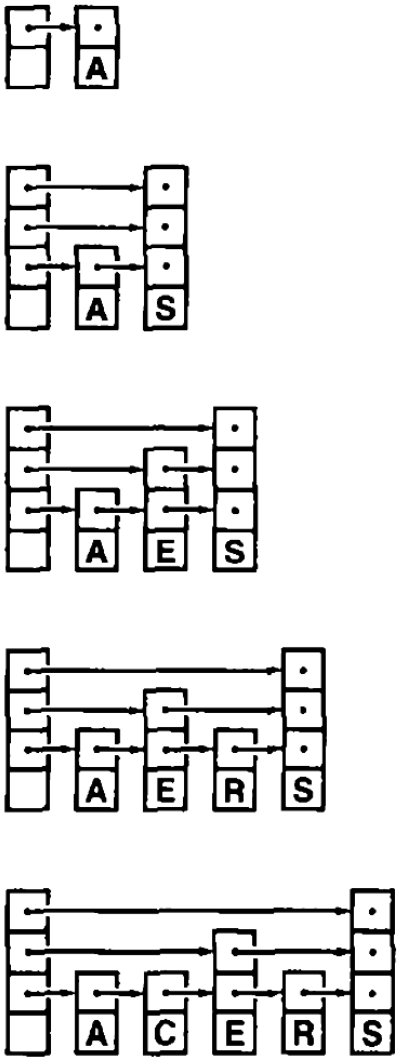
Notes

- The function `randX` on the previous slide generates a positive integer i with probability $1/2^i$.

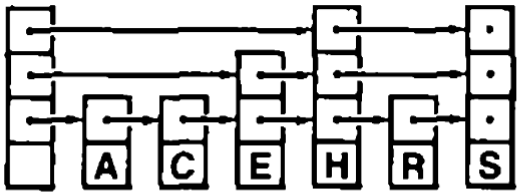
Example

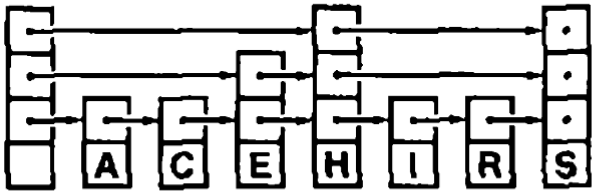
- The following slides show the construction of a skip list for a sample set of keys when inserted in random order.

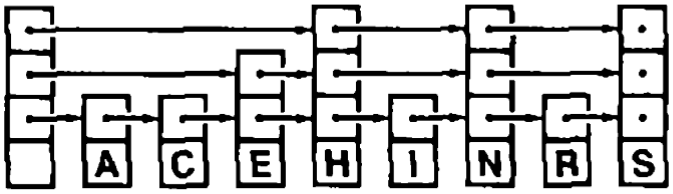
Example (cont'd)

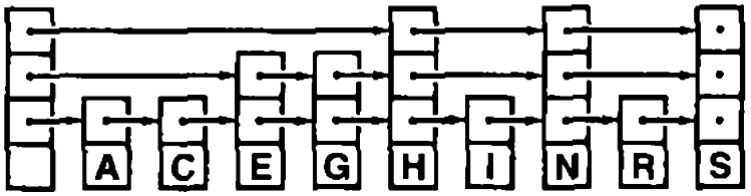


Example (cont'd)









Proposition

- Search and insertion in a randomized skip list with parameter t require about

$$\frac{(t \log_t N)}{2} = \frac{t}{2 \log_2 t} \log_2 N$$

comparisons, on the average.

- Proof omitted.

- **Note**: in the code presented earlier, we used $t = 2$.

Proposition

- Skip lists have $({}^{t}/_{t-1})N$ links on the average.
- Proof omitted.

Deletion in skip lists

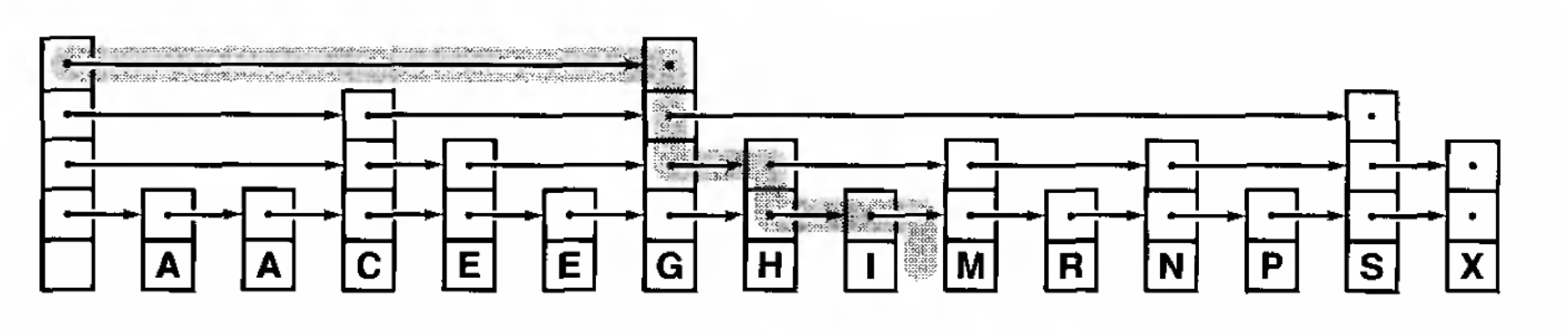
- The next slide presents an implementation of the delete function, using the same recursive scheme that we used for insert.
- To delete, we unlink the node from the lists at each level (where we linked it during insertion), and we free the node after unlinking it from the bottom list (as opposed to creating it before traversing the link for insert).

Deletion in skip lists (cont'd)

```
void deleteR(link t, Key v, int k)
  { link x = t->next[k];
    if (!less(key(x->item), v))
      {
        if (eq(v, key(x->item)))
          { t->next[k] = x->next[k]; }
        if (k == 0) { free(x); return; }
        deleteR(t, v, k-1); return;
      }
    deleteR(t->next[k], v, k);
  }

void STdelete(Key v)
     { deleteR(head, v, lgN); N--; }
```

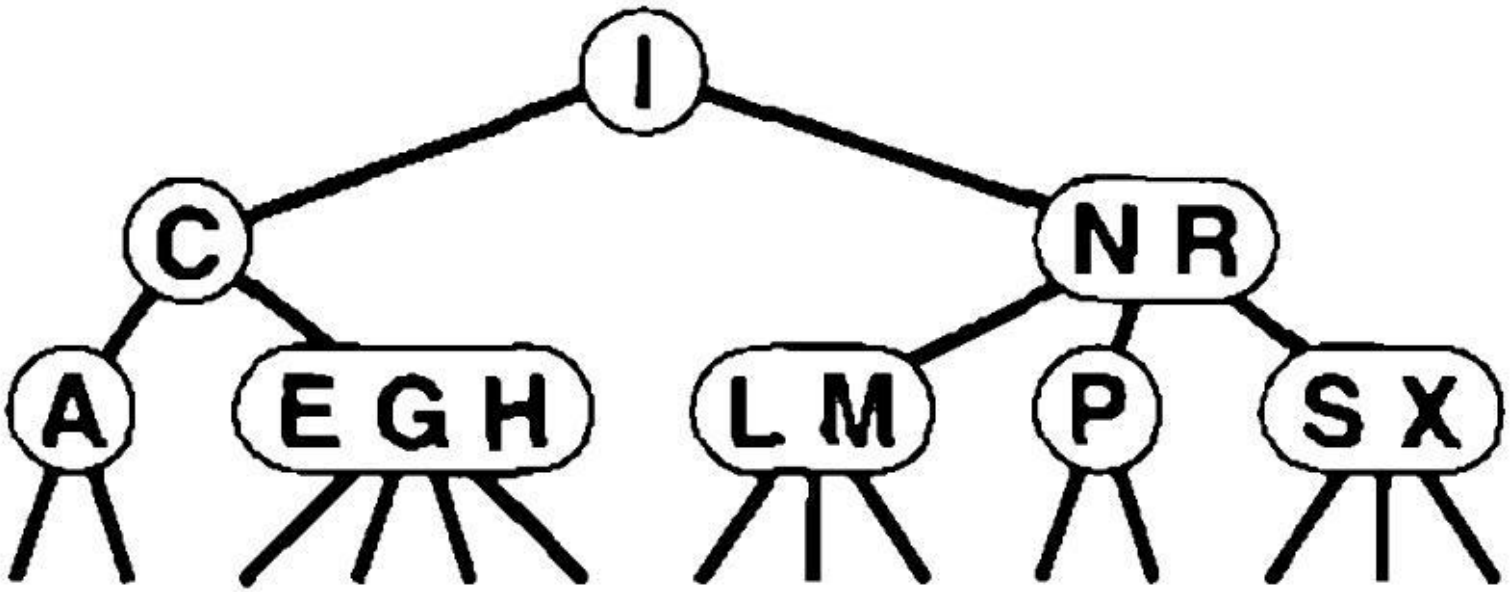
Example: delete H



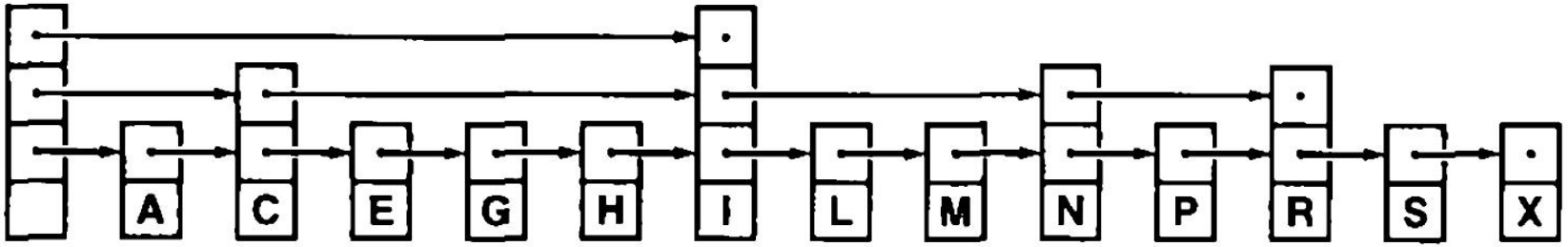
Skip lists vs. (2,4) trees

- Although skip lists are easy to conceptualize as a systematic way to move quickly through a linked list, it is also important to understand that the underlying data structure is nothing more than an alternative representation of a balanced tree.
- For example, the next two slides show a (2,4) tree and an equivalent skip list representation.

(2,4) tree



An equivalent skip list



Readings

- The material in the present slides comes verbatim from the following source:
  - R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.
    - Κεφάλαιο 13.5