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

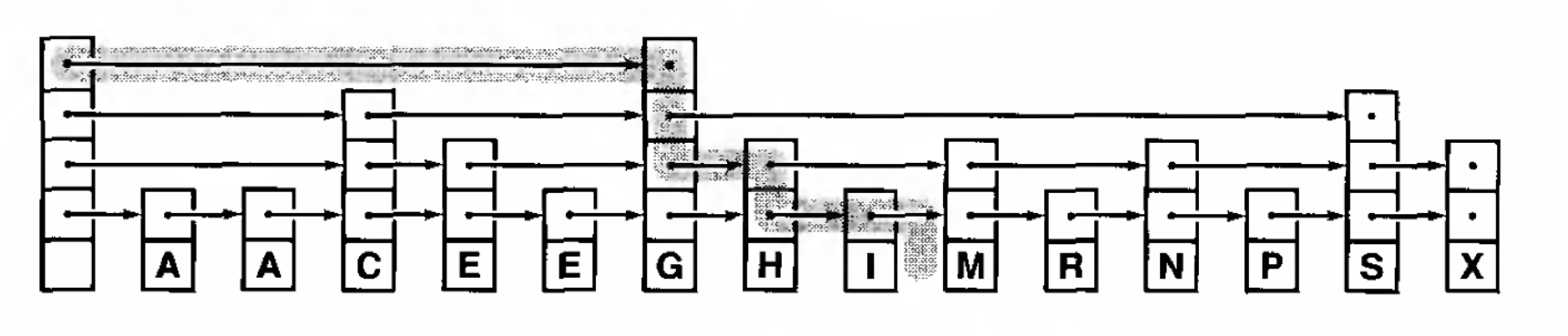
Notes

- 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.** These links are called **forward links** or **forward pointers.**
- **We can use the extra links to speed up search** as follows.
- **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 than the key we are looking for.
- **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 (forward pointers), 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 (forward pointers**). In other words, each node in a skip list has an array of next pointers—`next[0], next[1], ..., next[k]`. Each `next[i]` represents a forward link at level `i`.
- The element `sz` of a skip list node is the **number of links** in the node (depicted as **arrows or dots** in the figures). This is called the number of **forward levels**.
- The field `item` stores **the item of the node** of the list.
- The variable `N` keeps the **number of items in the list.**
- The variable `lgN` is the **current maximum number of forward levels in a node of the skip list.**
- `z` is a **sentinel node** (we will see below how it is used). `z` is not shown in the figures. Instead, we see a NULL pointer (a dot).

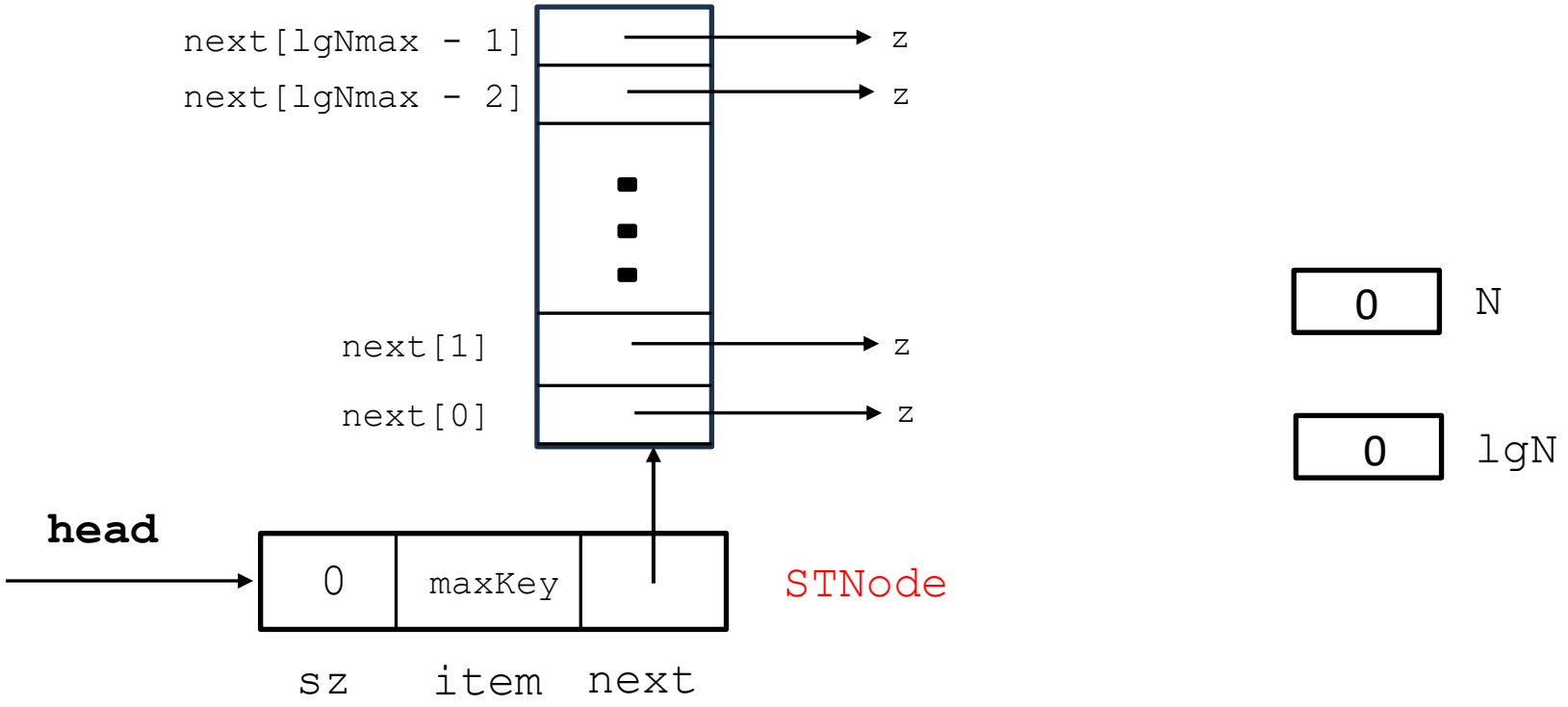
Skip list initialization

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

Skip list initialization (cont'd)

```
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;
  }
```

Skip list initialization (cont'd)



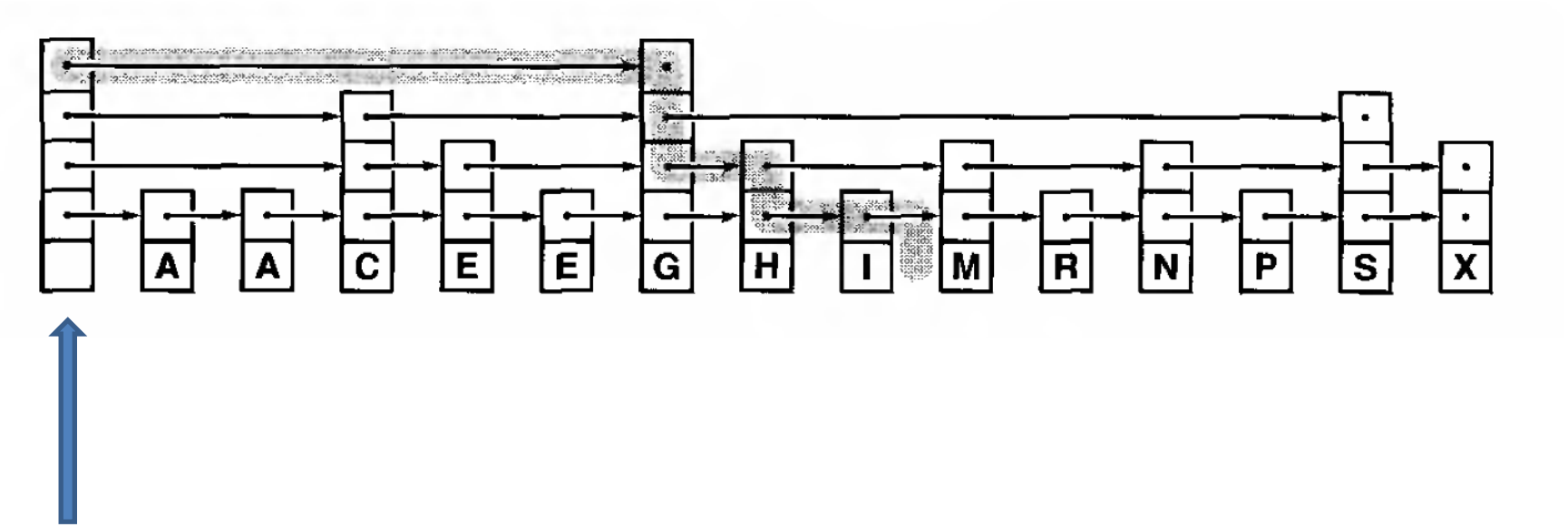
Notes

- 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 forward 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` **.**
- For simplicity, we assume that **items and keys are the same thing and both of them are integers.**
- The sentinel node `z` has item/key `maxKey` which is **larger than all keys in the list** (see later how it is used in search).

Notes (cont'd)

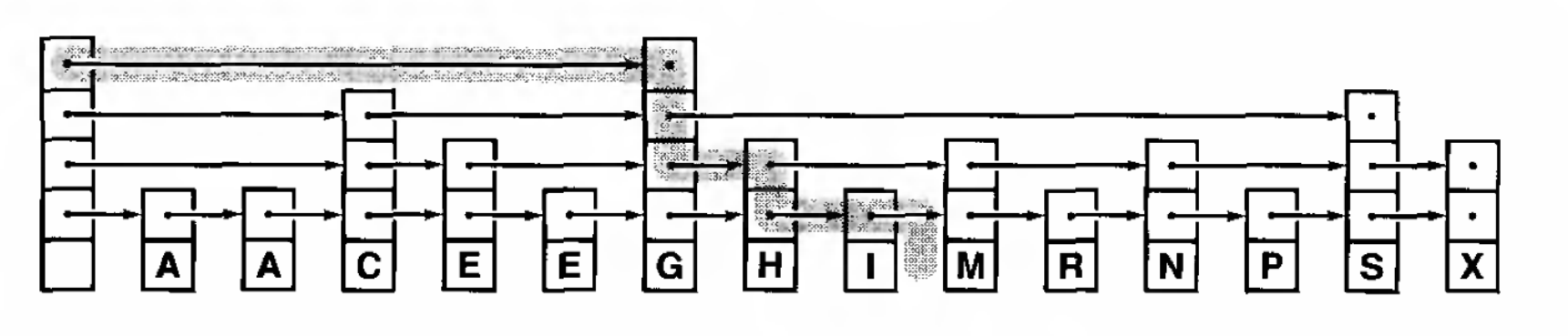
- Important: Function `NEW` allocates **two separate memory blocks:**
  - One for the `STnode` struct itself via the statement `link x=malloc(sizeof *x);`
  - One for the dynamic array next via the statement `x->next=malloc(k*sizeof(link));`
- **We have to be careful therefore when we delete a node** so that space is freed appropriately.

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 starting from the header node until we find the search key or a node that has a link to a next node with a larger key.**
- Then, **we move to the second-from-the-top list at the same node and iterate the same procedure.**
- **If the next node has a key smaller than the search key then 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 `randX` 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 $i-1$, **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 $i-1$) 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 second argument `NEW(v, randX())` i.e., a node created with function `NEW` which has 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 start counting at 0), 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`.
- **Moving down a level is done by the recursive call with third argument `k-1`** (until `k` becomes `0`).

The function `randX`

- Now we have to define the function `randX` of the previous slide so that it generates a positive integer i with probability $1/2^i$.
- `randX` **introduces randomness into the data structure**, **which ensures good average-case performance** as we will see later.

The function `randX` (cont'd)

```
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;
   }
```

- Remember:
  - The constant `lgNmax` is the **maximum number of levels that we allow in the list.**
  - The variable `lgN` is the **current maximum number of levels in a node of the skip list.**

The function `randX` (cont'd)

- Let us explain the details of `randX`.
- First `i, j` and `t` are defined, and `t` is assigned a **pseudo-random integer** in the interval `[0, RAND_MAX]` (this is done by the call to `rand()`).
- This means that **`t` is uniformly distributed over that interval.**

What Are Pseudo-Random Numbers?

- **Pseudo-random numbers** are numbers that **appear random, but are actually generated by a deterministic algorithm.**

Pseudo-random Numbers

- Pseudo-random numbers have the following properties:
  - **Generated by a formula or algorithm.** A pseudo-random number generator (PRNG) uses a math formula or algorithm to compute the next number in a sequence based on the previous one (starting from a seed).
  - **Deterministic.** If you start with the same seed, you always get the same sequence of numbers.
  - **Not truly random.** In contrast to true randomness (like radioactive decay or atmospheric noise), PRNGs are predictable if you know the seed and algorithm.

The function `randX` (cont'd)

- Let us now consider the code

```
for (i = 1, j = 2; i < lgNmax; i++, j += j)
        if (t > RAND_MAX/j) break;
```

- We want to assign a level i to the new node with probability $1/2^i$.

Level `i`	Probability
1	1/2
2	1/4
3	1/8
…	…
k	1/2^k

- This is exactly a **geometric distribution** (with success probability 1/2).

The function `randX` (cont'd)

- Let us now concentrate on the **execution of the `for` loop**.
- At level `i=1, j=2,` we have `RAND_MAX/j= RAND_MAX/2.` So, the loop breaks early if `t` is in the upper half of `[0, RAND_MAX].` That happens with probability ½.
- At level `i=2, j=4,` we have `RAND_MAX/4.` So, the loop breaks with probability ¾ , because `t > RAND_MAX/4` happens 75% of the time.
- **In general: For level `i`, the condition `t > RAND_MAX/j` becomes more likely as `j` increases (since the threshold `RAND_MAX/j` gets smaller).**
- **So, the loop breaks earlier more often, meaning lower levels are assigned more often than higher levels.**
- **Therefore,** `randX` generates a level i with probability $1/2^i$.

Why it Works

This distribution:

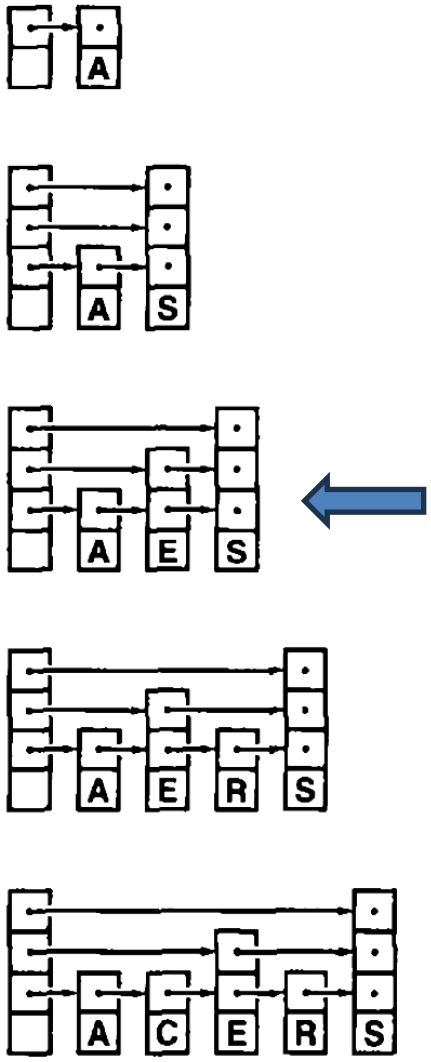
- Ensures most nodes have 1 or 2 levels.
- But a **few lucky nodes** will have many levels (acting like “express lanes”).
- This randomness **balances the skip list probabilistically**, similar to how balancing works in AVL or red-black trees—but without rotation logic.

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)

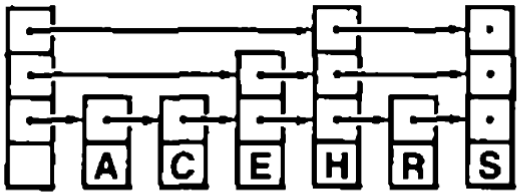
This first step can be a bit misleading. In the example, `lgNmax` is `3` but in the first list, the header node is shown with one forward pointer only. It should have been drawn with three forward pointers as in the second and subsequent steps.

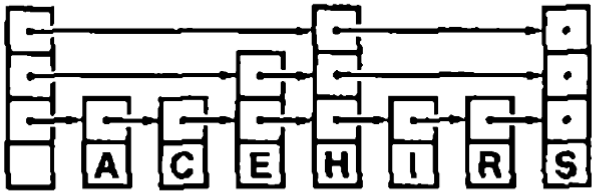


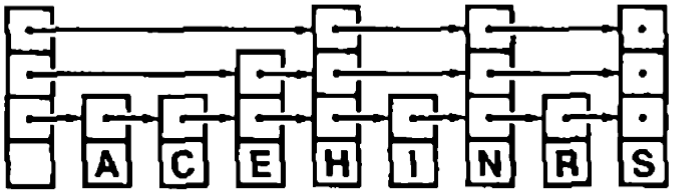
We assume here that in the case of inserting key E, `randX()` returned `2`. Therefore, `x->sz` is `2`. So, when we reach level `(x->sz)-1` (i.e., `1`), we link the new node with the next one and again when `k` reaches `0`.

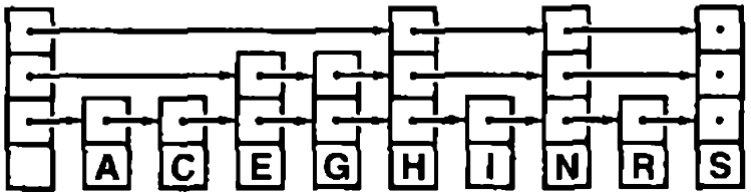
The remaining insertions are similar.

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**.
- This process involves:
  - **Unlinking the node from the lists at each level where we linked it during insertion.**
  - **Freeing the node after unlinking it from the bottom list** (as opposed to creating it with `NEW` 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->next); free(x); return; }
        deleteR(t, v, k-1); return;
      }
    deleteR(t->next[k], v, k);
  }

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

Deletion in skip lists (cont'd)

- The **unlinking** at each level is done by the statement

```
t->next[k] = x->next[k];
```

- Freeing the space occupied by the node needs two `free` statements:
  - One for the dynamic array of forward pointers (`free(x->next);`)
  - One for the `STNode` struct (`free(x);`)

Deletion in skip lists (cont'd)

- Note also the use of `!less(...)` in `delete`. It ensures that:
  - We don't keep skipping past the node we want to delete.
  - We stop if the next node has a key equal to or greater than the key to delete.
  - Only if it's exactly equal, we delete it.

Deletion in skip lists (cont'd)

- Finally, note that the version of deletion we presented does not check for the case that the key to be deleted is not in the list.
- The function on the next slide solves this issue.

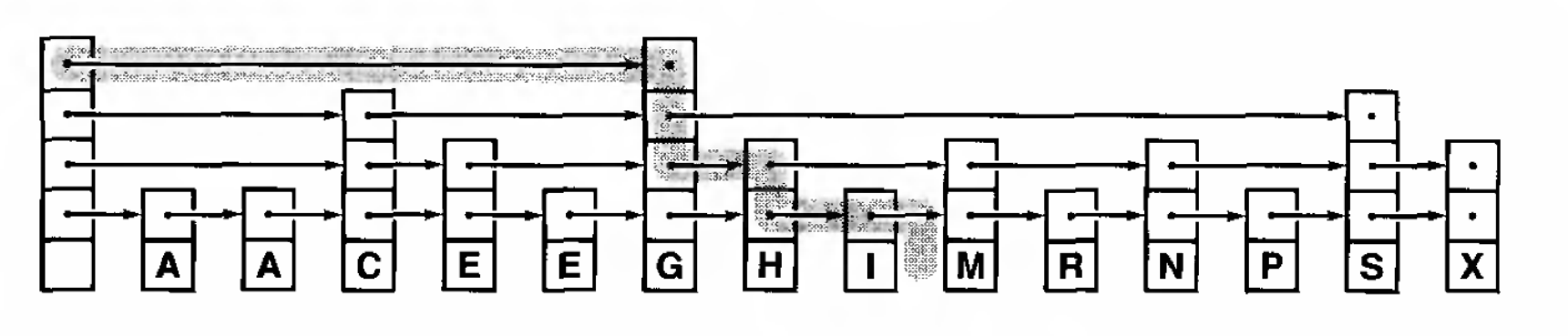
Deletion in skip lists (cont'd)

```
// Public interface that adjusts N only if delete
succeeded
void STdelete(Key v) {
    if (deleteR(head, v, lgN)) {
        N--;
        printf("Key %d deleted successfully.\n", v);
    } else {
        printf("Key %d not found. No deletion
performed.\n", v);
    }
}
```

Deletion in skip lists (cont'd)

```
// Recursive delete now returns 1 if deletion occurred, 0 otherwise
int deleteR(link t, Key v, int k) {
    link x = t->next[k];
    if (!less(key(x->item), v)) { // i.e., if key(x) >= v
       if (eq(v, key(x->item))) {
           t->next[k] = x->next[k]; // unlink at this level
           if (k == 0) {
              free(x->next);       // clean up memory
              free(x);
              return 1;           //  deletion successful
           }
       }
       return deleteR(t, v, k - 1); // go down a level
    }
    return deleteR(t->next[k], v, k); // move forward
}
```

Example: delete H



Please ignore the gray shading after node H. We have found H, so we don't need to move further.

Complete program

- Let us now write a complete C program that implements linked lists and has a `main` that:
  - Inserts the keys 10, 12, 7, 8 and 9 in the list
  - Searches for keys 7 and 8 (does not exist).
  - Deletes the keys 7 and 8 (does not exist!) in the list.
  - Prints appropriate messages in each case.

Complete program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define lgNmax 10
#define NULLitem -1
#define maxKey 9999

typedef int Item;
typedef int Key;

typedef struct STnode *link;

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

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

Complete program (cont'd)

```
Key key(Item item) {
    return item;
}

int eq(Key a, Key b) {
    return a == b;
}

int less(Key a, Key b) {
    return a < b;
}
```

Complete program (cont'd)

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

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

Complete program (cont'd)

```
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;
}
```

Complete program (cont'd)

```
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);
}
```

Complete program (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++;
}
```

Complete program (cont'd)

```
// Safe delete: only free and decrement N if item is found
int 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->next);
                free(x);
                return 1; // successfully deleted
            }
        }
        return deleteR(t, v, k - 1);
    }
    return deleteR(t->next[k], v, k);
}

void STdelete(Key v) {
    if (deleteR(head, v, lgN)) {
        N--;
        printf("Key %d deleted successfully.\n", v);
    } else {
        printf("Key %d not found. No deletion performed.\n", v);
    }
}
```

Complete program (cont'd)

```
// Main demo
int main() {
    srand(time(NULL));
    STinit(lgNmax);

    int keys[] = {10, 12, 7, 8, 9};

    printf("Inserting keys:\n");
    for (int i = 0; i < 5; i++) {
        printf("- %d\n", keys[i]);
        STinsert(keys[i]);
    }

    printf("\nSearching for key 7:\n");
    if (STsearch(7) != NULLitem)
        printf("Key 7 found in skip list.\n");
    else
        printf("Key 7 not found in skip list.\n");

    printf("\nSearching for key 8:\n");
    if (STsearch(8) != NULLitem)
        printf("Key 8 found in skip list.\n");
    else
        printf("Key 8 not found in skip list.\n");

    printf("\nDeleting key 7:\n");
    STdelete(7);

    printf("Deleting key 8:\n");
    STdelete(8);

    return 0;
}
```

`srand(time(NULL));`

- Notice the above statement in the `main` function.
- What does it do?

`rand()` and Reproducibility

- In C, `rand()` generates **pseudo-random numbers.**
- By default, if you call `rand()` without doing anything else, **you'll get the same sequence of numbers every time your program runs**.
- This happens because `rand()` uses a **seed value** internally to start its sequence — and if you don’t set it, it uses the same default seed every time.

What `srand()` Does

- `srand()` **sets the seed** for the `rand()` function.
- Think of the seed as the "starting point" of the random number sequence.
- If you set the same seed, you'll get the same sequence of random numbers.

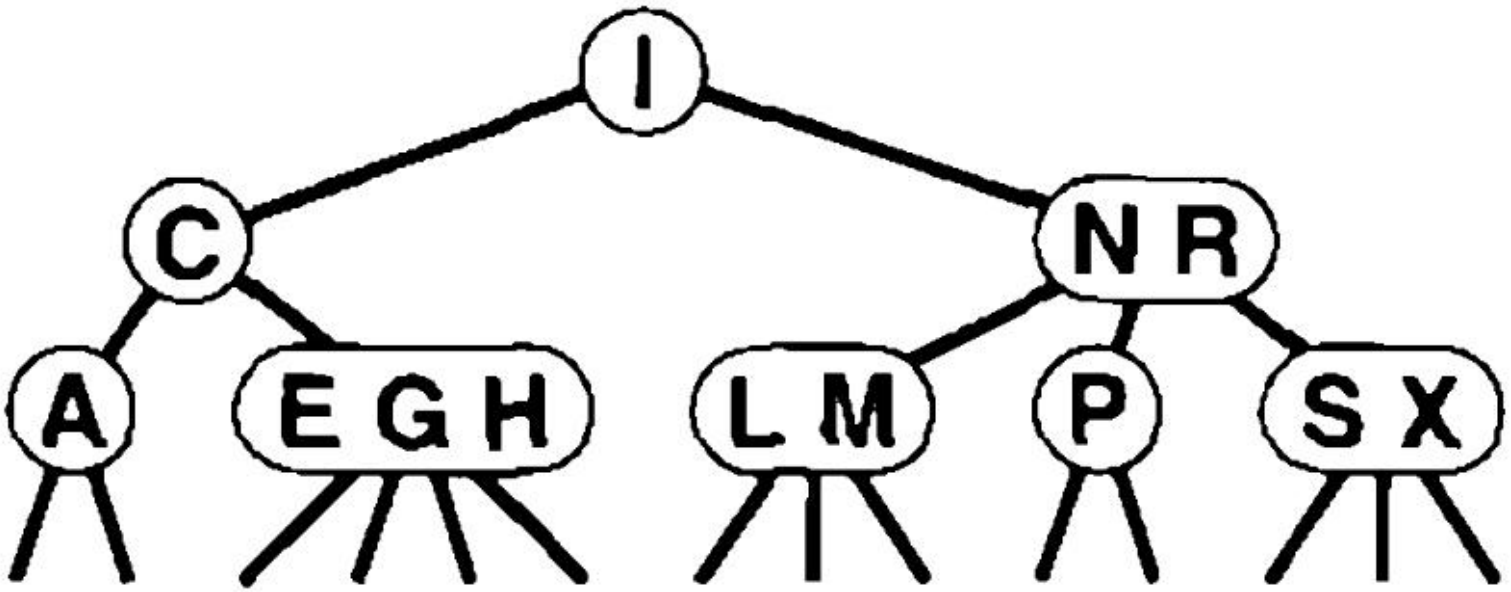
So Why `time(NULL)`?

- `time(NULL)` returns the current time in seconds since the Unix epoch (January 1, 1970).
- **This value is (almost) always different each time you run the program.**
- This makes your skip list insertions (and random levels) **unpredictable and different on each run**, which is ideal for simulating real-world behavior.

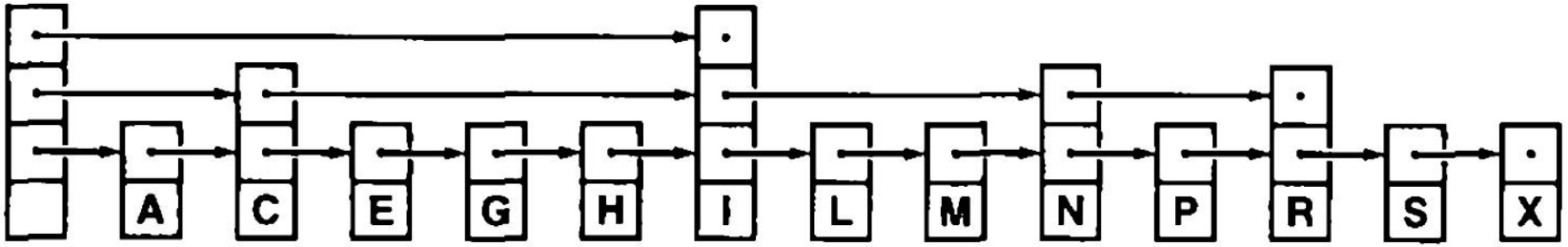
Skip lists vs. (2,4) trees

- 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