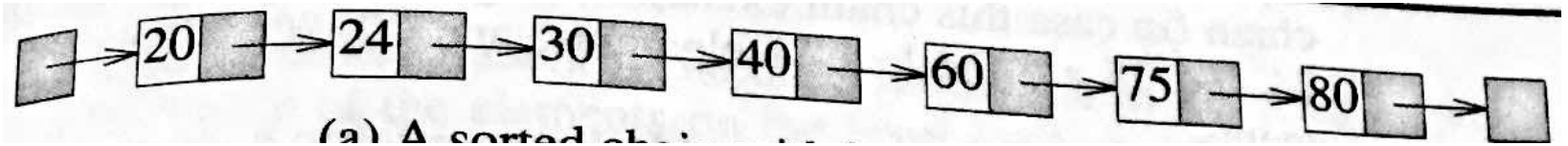
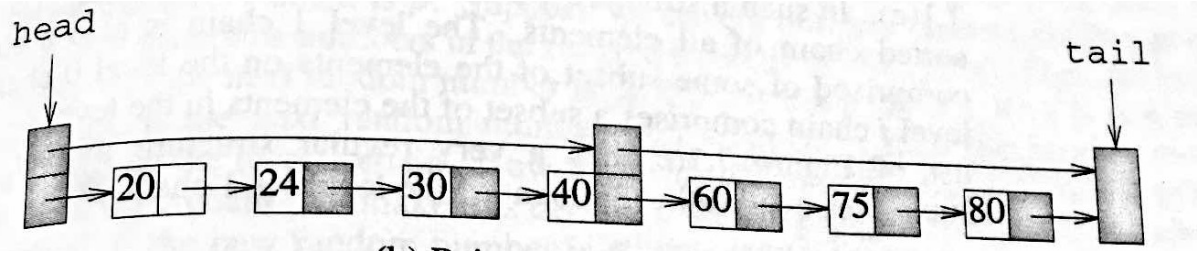
Λίστες Παράλειψης

Βασική Ιδέα

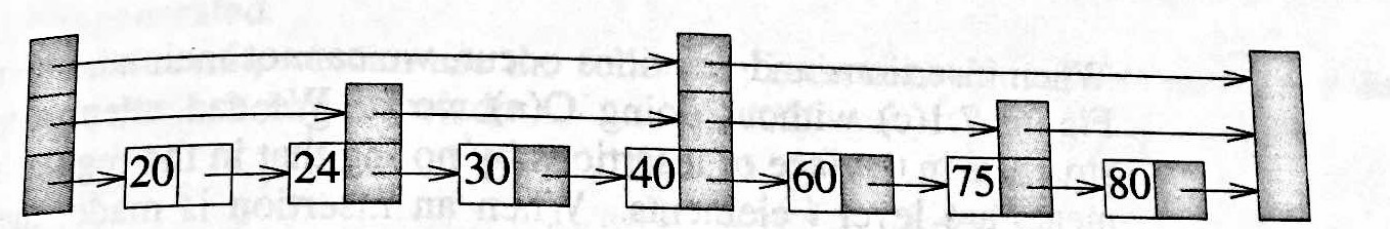
- Σε μία ταξινομημένη αλυσίδα n στοιχείων, χρόνος χειρότερης περίπτωσης αναζήτησης: n συγκρίσεις.



Αν υπάρχουν δείκτης στο μεσαίο στοιχείο: μείωση στο μισό του απαιτούμενου χρόνου



Ομοίως αν υπάρχουν δείκτες στα μεσαία στοιχεία των δύο ισομηκών τμημάτων του πίνακα: μείωση στο 1/4 του απαιτούμενου χρόνου



Γενική περίπτωση

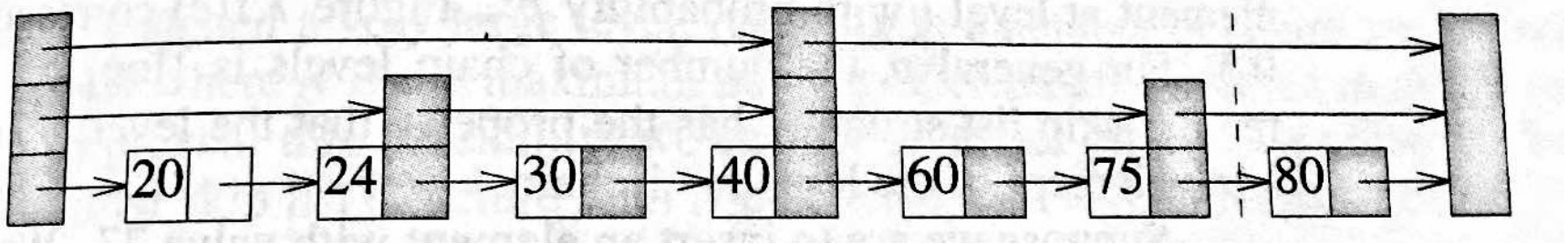
- Λίστα Παράλειψης:
  - Ιεραρχία αλυσίδων:
    - Επίπεδο 0: ταξινομημένη αλυσίδα με όλα τα στοιχεία
    - Επίπεδο 1: ταξινομημένη αλυσίδα που περιέχει υποσύνολο των στοιχείων του Επιπέδου 1

    …..

    - Επίπεδο i: ταξινομημένη αλυσίδα που περιέχει υποσύνολο των στοιχείων του Επιπέδου i-1

Εισαγωγή στοιχείου

- Εισαγωγή του 77
- Αναζήτηση πρώτα του στοιχείο μέσα στη λίστα παράλειψης:



- Αποθηκεύουμε όλους τους προηγούμενους κόμβους σε όλα τα επίπεδα από αυτή την εισαγωγή.

Εισαγωγή στοιχείου (Συν.)

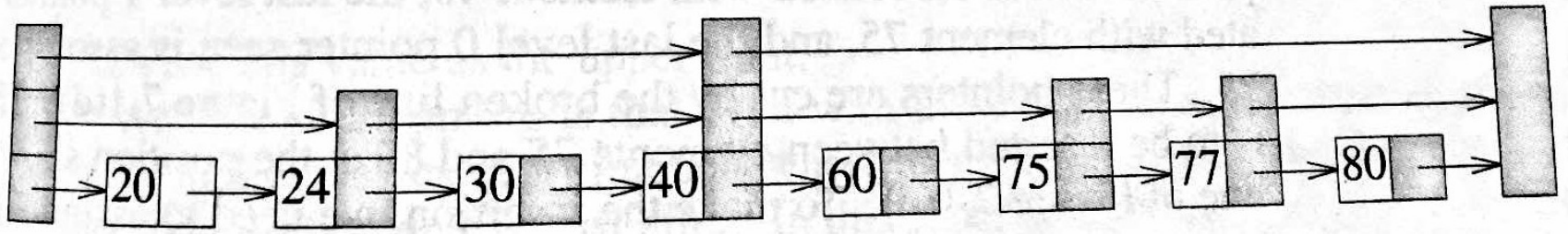
- Καθορισμός του πλήθους των επιπέδων που θα συμμετάσχει το νέο στοιχείο.
- Συμμετέχει σε συνεχόμενα επίπεδα με αρχή το επίπεδο 0.
- p: πιθανότητα το στοιχείο να συμμετάσχει στο επίπεδο i δεδομένου ότι συμμετείχε στο επίπεδο i-1.
- rand(): Συνάρτηση που παράγει αριθμούς από 0 μέχρι RAND_MAX
- Ο αλγόριθμος προσδιορισμού του πλήθους των επιπέδων:

```
Cutoff = p*RAND_MAX
lev=0;
while (rand() <= Cutoff) lev++
```

- Το νέο στοιχείο θα συμμετέχει στα επίπεδα 0,1,…,level.
- Το αναμενόμενο πλήθος επιπέδων σε μία λίστα παράλειψης με N στοιχεία είναι $\lceil log_{1/p} N \rceil - 1$

Εισαγωγή στοιχείου (Συν.)

- Εισαγωγή κόμβων στα αντίστοιχα επίπεδα.
- Προσαρμογή των δεικτών των κόμβων που είναι πριν το σημείο εισαγωγής.
- Εισαγωγή του 77.



Διαγραφή στοιχείου

- Αναζήτηση του στοιχείου
- Αφαίρεση των αντίστοιχων κόμβων από τα επίπεδα που συμμετέχει το στοιχείο.

```
template<class E, class K>
class SkipNode {
   friend SkipList<E,K>;
   private:
      SkipNode(int size)
         {link = new SkipNode<E,K> *[size];}
      ~SkipNode() {delete [] link;}
      E data;
      SkipNode<E,K> **link;  // 1D array of pointers
};
```

Πρόγραμμα 7.4 Κλάση `SkipNode`

```
template<class E, class K>
class SkipList {
   public:
      SkipList(K Large, int MaxE = 10000,
                        float p = 0.5);
      ~SkipList();
      bool Search(const K& k, E& e) const;
      SkipList<E,K>& Insert(const E& e);
      SkipList<E,K>& Delete(const K& k, E& e);
   private:
      int Level();
      SkipNode<E,K> *SaveSearch(const K& k);
      int MaxLevel;   // max permissible chain level
      int Levels;     // max current nonempty chain
      int CutOff;     // used to decide level number
      K TailKey;      // a large key
      SkipNode<E,K> *head;  // head node pointer
      SkipNode<E,K> *tail;  // tail node pointer
      SkipNode<E,K> **last; // array of pointers
};
```

Πρόγραμμα 7.5 Κλάση SkipList

```
template<class E, class K>
SkipList<E,K>::SkipList(K Large, int MaxE, float p)
{// Constructor.
   CutOff = p * RAND_MAX;
   MaxLevel = ceil(log(MaxE) / log(1/p)) - 1;
   TailKey = Large;
   randomize(); // initialize random generator
   Levels = 0;  // initial number of levels

   // create head & tail nodes and last array
   head = new SkipNode<E,K> (MaxLevel+1);
   tail = new SkipNode<E,K> (0);
   last = new SkipNode<E,K> *[MaxLevel+1];
   tail->data = Large;

   // head points to tail at all levels as empty
   for (int i = 0; i <= MaxLevel; i++)
      head->link[i] = tail;
}
```

```
template<class E, class K>
SkipList<E,K>::~SkipList()
{// Delete all nodes and array last.
   SkipNode<E,K> *next;

   // delete all nodes by deleting level 0
   while (head != tail) {
      next = head->link[0];
      delete head;
      head = next;
      }
   delete tail;

   delete [] last;
}
```

Πρόγραμμα 7.6 Μέθοδοι κατασκευής και καταστροφής

```
class element {
   friend void main(void);
   public:
      operator long() const {return key;}
      element& operator =(long y)
      {key = y; return *this;}
   private:
      int data;
      long key;
};
```

Πρόγραμμα 7.7 Υπερφόρτωση τελεστή για λίστες παράλειψης

```
template<class E, class K>
bool SkipList<E,K>::Search(const K& k, E& e) const
{// Search for element that matches k.
 // Put matching element in e.
 // Return false if no match.
   if (k >= TailKey) return false;

 // position p just before possible node with k
 SkipNode<E,K> *p = head;
 for (int i = Levels; i >= 0; i--) // go down levels
    while (p->link[i]->data < k)   // follow level i
       p = p->link[i];             // pointers

// check if next node has key k
e = p->link[0]->data;
return (e == k);
}
```

```
template<class E, class K>
SkipNode<E,K> * SkipList<E,K>::SaveSearch(const K& k)
{// Search for k and save last position
 // visited at each level.
   // position p just before possible node with k
   SkipNode<E,K> *p = head;
   for (int i = Levels; i >= 0; i--) {
      while (p->link[i]->data < k)
         p = p->link[i];
      last[i] = p;  // last level i node seen
      }
   return (p->link[0]);
}
```

Πρόγραμμα 7.8 Συναρτήσεις αναζήτησης σε λίστες παράλειψης

```
template<class E, class K>
int SkipList<E,K>::Level()
{// Generate a random level number <= MaxLevel.
   int lev = 0;
   while (rand() <= CutOff)
      lev++;
   return (lev <= MaxLevel) ? lev : MaxLevel;
}

template<class E, class K>
SkipList<E,K>& SkipList<E,K>::Insert(const E& e)
{// Insert e if not duplicate.
   K k = e; // extract key
   if (k >= TailKey) throw BadInput(); // too large

   // see if duplicate
   SkipNode<E,K> *p = SaveSearch(k);
   if (p->data == e) throw BadInput(); // duplicate
```

```
    // not duplicate, determine level for new node
    int lev = Level(); // level of new node
    // fix lev to be <= Levels + 1
    if (lev > Levels) {lev = ++Levels;
                       last[lev] = head;}

    // get and insert new node just after p
    SkipNode<E,K> *y = new SkipNode<E,K> (lev+1);
    y->data = e;
    for (int i = 0; i <= lev; i++) {
      // insert into level i chain
      y->link[i] = last[i]->link[i];
      last[i]->link[i] = y;
      }

  return *this;
}
```

Πρόγραμμα 7.9 Εισαγωγή σε λίστα παράλειψης

```
template<class E, class K>
SkipList<E,K>& SkipList<E,K>::Delete(const K& k, E& e)
{// Delete element that matches k.  Put deleted
 // element in e.  Throw BadInput if no match.
   if (k >= TailKey) throw BadInput(); // too large

   // see if matching element present
   SkipNode<E,K> *p = SaveSearch(k);
   if (p->data != k) throw BadInput(); // not present

   // delete node from skip list
   for (int i = 0; i <= Levels &&
                      last[i]->link[i] == p; i++)
      last[i]->link[i] = p->link[i];

   // update Levels
   while (Levels > 0 && head->link[Levels] == tail)
      Levels--;

   e = p->data;
   delete p;
   return *this;
}
```

Πρόγραμμα 7.10 Διαγραφή από λίστα παράλειψης

Πολυπλοκότητα

- Πολυπλοκότητα Χρόνου:
  - Αναζήτηση, Εισαγωγή, Διαγραφή: O(n+MaxLevel)
  - Αναμενόμενη πολυπλοκότητα: O(logn)
- Πολυπλοκότητα Χώρου:
  - Θ(n/(1-p))