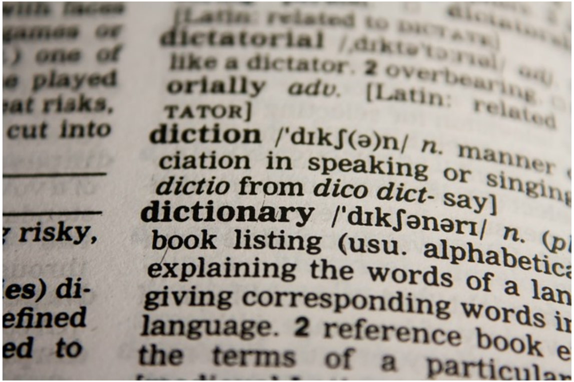
Computer Science 3A - CSC3A10

Lecture 9: Dictionary and Skip Lists

Academy of Computer Science and Software Engineering
University of Johannesburg

Dictionary ADT



Dictionary ADT Properties

- The dictionary ADT models a search-able collection of key-value pairs (k,v) which we call entries
- The main operations of a dictionary are searching, inserting, and deleting items
- Multiple items with the same key are allowed
- Applications:
  - word-definition pairs
  - credit card authorizations
  - DNS mapping of host names (eve.uj.ac.za) to Internet IP addresses (e.g., 152.106,120.2)

Dictionary ADT Methods

- **find(k)**: if the dictionary has an entry with key k, returns it, else, returns null
- **findAll(k)**: returns an iterator of all entries with key k
- **insert(k, o)**: inserts and returns the entry (k, o)
- **remove(e)**: remove the entry e from the dictionary
- **entries()**: returns an iterator of the entries in the dictionary
- **size(), isEmpty()**

Dictionary ADT Example

Operation	*Output*	*Dictionary*
insert(5,A)	(5,A)	(5,A)
insert(7,B)	(7,B)	(5,A),(7,B)
insert(2,C)	(2,C)	(5,A),(7,B),(2,C)
insert(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
insert(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(4)	null	(5,A),(7,B),(2,C),(8,D),(2,E)
find(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
findAll(2)	(2,C),(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
remove(find(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
find(5)	null	(7,B),(2,C),(8,D),(2,E)

A List-Based Dictionary

- A log file or audit trail is a dictionary implemented by means of an unsorted sequence. We store the items of the dictionary in a sequence (based on a doubly-linked list or array), in arbitrary order
- **Performance**:
    - *insert* takes O(1) time since we can insert the new item at the beginning or at the end of the sequence
    - *find* and *remove* take O(n) time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

A List-Based Dictionary II

- The log file is effective only for dictionaries of small size or for dictionaries on which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)
- Therefore, very much to their ADT's this constrains, the potential applications it can be used in!

List-Based Dictionary Algorithms

```
Algorithm findAll(k):
Input: A key k
Output: An iterator of entries with key equal to k
Create an initially empty list L
for each entry e in D.entries do
  if e.key() = k  then
        L.addLast(e)
return L.elements()
```

findAll

List-Based Dictionary Algorithms II

```
Algorithm insert(k,v):
Input: A key k and value v
Output: The entry (k,v) added to D
Create a new entry e = (k,v)
S.addLast(e)  {S is unordered}
return e
```

insert

List-Based Dictionary Algorithms III

```
Algorithm remove(e):
Input: An entry e
Output: The removed entry e or null if e was not in D
{We don't assume here that e stores its location in S}
For each entry p in S.positions do
  if p.element() = e then
      S.remove(p)
  return e
return null   {there is no entry e in D}
```

remove

Hash Table Implementation

- We can also create a hash-table dictionary implementation.
- If we use separate chaining to handle collisions, then each operation can be delegated to a list-based dictionary stored at each hash table cell.

Hash Table Implementation

```
Algorithm insert(k,v):
Input: A key k and a value v
Output: The entry (k,v) added to D
if (n+1)/N > alpha then
  Double the size of A and rehash all the existing entries
e=A[h(k)].insert (k,v)
n=n+1
return e
```

insert

Hash Table Implementation II

```
Algorithm findAll(k):
Input: A key k
Output: An Iterable collection of entries with key equal to k
return A[h(k)].findAll(k)
```

findAll

Hash Table Implementation III

```
Algorithm remove(e):
Input:  An entry e
Output: The removed entry e
t = A[h(k)].remove(e)
n = n - 1
return t
```

remove

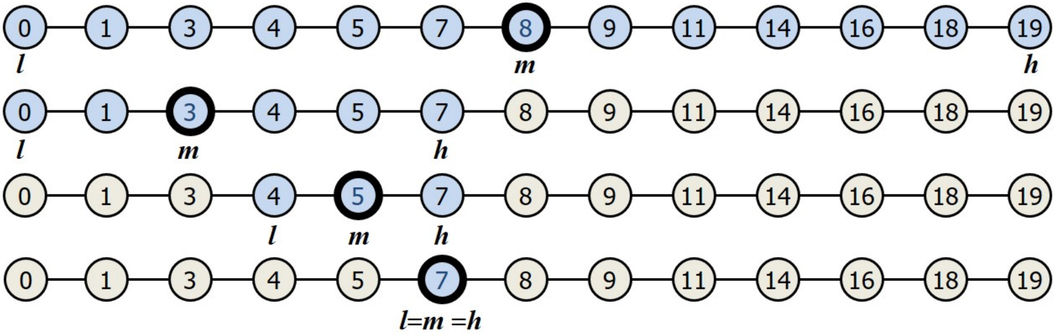
Dictionary Binary Search

Binary search performs operation find(k) on a dictionary implemented by means of an array-based sequence, sorted by key

- similar to the high-low game
- at each step, the number of candidate items is halved
- terminates after a logarithmic number of steps

Dictionary Binary Search II

Example: find(7)



Array-Based Dictionary

A search table is a dictionary implemented by means of a sorted array

- We store the items of the dictionary in an array-based sequence, sorted by key
- We use an external comparator for the keys

Performance:

- **find** takes O(log n) time, using binary search
- **insert** takes O(n) time since in the worst case we have to shift n/2 items to make room for the new item
- **remove** takes O(n) time since in the worst case we have to shift n/2 items to compact the items after the removal

Search Table II

- A search table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)
- Thereby, also limiting it potential use.

Skip List ADT

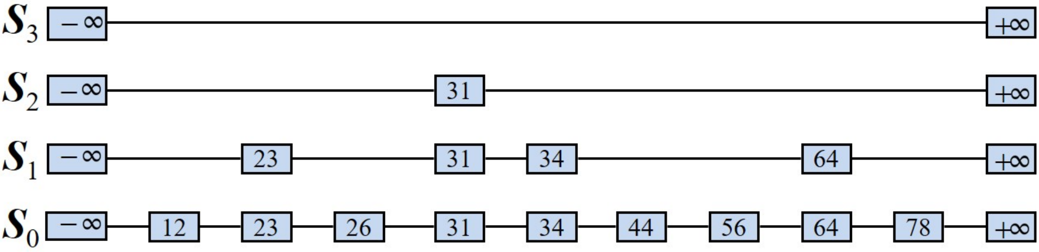
Skip List Properties

A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that:

- Each list S_i contains the special keys $+\infty$ and $-\infty$
- List S_0 contains the keys of S in nondecreasing order
- Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
- List S_h contains only the two special keys

We show how to use a skip list to implement the dictionary ADT

Skip List Example



Skip List Algorithm

```
Algorithm SkipSearch(k):
Input: A search key k
Output: Position p in the bottom list S0 such that the entry p has the
    largest key less than or equal to k
p = s
while below(p) != null do
   p = below(p)   //drop down
   while k >= key(next(p)) do
            p = next(p) // scan forward
return p
```

SkipSearch

Skip List Algorithm II

```
Algorithm SkipInsert(k,v):
Input: A key k and a value v
Output: The entry (k,v) inserted into the skip list added to D
// insertAfterAbove(p,q,(k,v)) insert a position storing (k,v) after
// position p (on the same level as p) and above q and returning
     position r the new entry
p = SkipSearch(k)
q = insertAfterAbove(p,null,(k,v)) //we are at the bottom level.
e = q.element
i = 0
```

SkipInsert

Skip List Algorithm III

```
while coinflip() = heads do
  i = i+1
  if i >= h then
      h = h +1  // add a new level to the skip list
      t = next(s)
      s = insertAfterAbove(null,s,(-INFINTY,null))
      insertAfterAbove(s,t,(+INFINITY,null))
  while above(p) = null do
      p = prev(p)  //scan backwards
  p =above(p) //jump up to upper level
  q = insertAfterAbove(p,q,e) // add a position to the tower of the new
      entry
n = n+1
return e
```

SkipInsert (Continued)

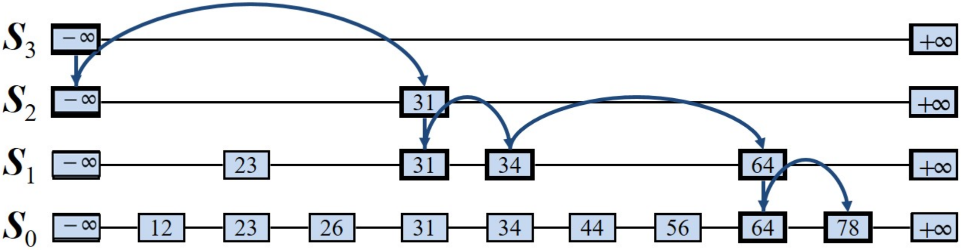
Skip List Search

We search for a key x in a a skip list as follows:

- We start at the first position of the top list
- At the current position p, we compare x with y = key(next(p))
  - $x = y$: we return element(next(p))
  - $x > y$: we "scan forward"
  - $x < y$: we "drop down"
- If we try to drop down past the bottom list, we return *null*

Skip Search Search II

Example: search for 78



12

Randomized Algorithms

A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution

```
b = Random()
  if  b = 0
    do A ...
  else { b = 1}
    do  B ...
```

Random Algorithm Structure

Randomized Algorithms II

- Its running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions:
  - the coins are unbiased, and
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")
- We use a randomized algorithm to insert items into a skip list

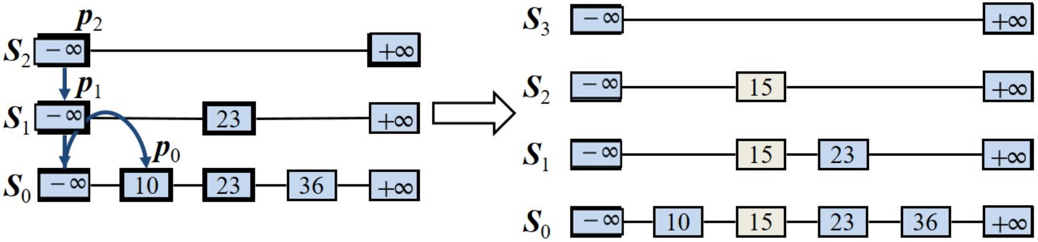
Skip List Insertion

To insert an entry (x, o) into a skip list, we use a randomized algorithm:

- We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
- If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
- We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
- For j = 0, ..., i, we insert item (x, o) into list S_j after position p_j

Skip List Insertion II

Example: insert key 15, with i = 2



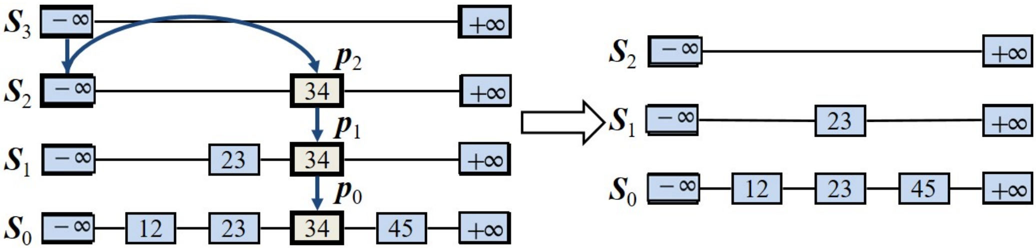
Skip List Removal

To remove an entry with key x from a skip list, we proceed as follows:

- We search for x in the skip list and find the positions $p_0, p_1, ..., p_i$ of
- the items with key x, where position pj is in list S_j
- We remove positions $p_0, p_1, ..., p_i$ from the lists $S_0, S_1, ..., S_i$
- We remove all but one list containing only the two special keys

Skip List Removal II

Example: remove key 34

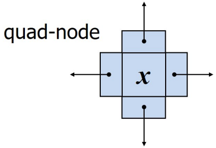


Skip List Implementation

We can implement a skip list with quad-nodes.
A quad-node stores:

- entry
- link to the node prev
- link to the node next
- link to the node below
- link to the node above

Also, we define special keys *PLUS_INF* and *MINUS_INF*, and we modify the key comparator to handle them



Space Usage

The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm. We use the following two basic probabilistic facts:

- **Fact 1**: The probability of getting i consecutive heads when flipping a coin is $1/2^i$
- **Fact 2**: If each of n entries is present in a set with probability p, the expected size of the set is np

Space Usage II

Consider a skip list with n entries

- By Fact 1, we insert an entry in list S_i with probability $1/2^i$
- By Fact 2, the expected size of list S_i is $n/2^i$

Space Usage III

The expected number of nodes used by the skip list is

$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n \tag{1}$$

Thus, the expected space usage of a skip list with n items is $O(n)$

Height

- The running time of the search and insertion algorithms is affected by the height h of the skip list
- We show that with high probability, a skip list with n items has height $O(logn)$
- We use the following additional probabilistic fact:
- **Fact 3**: If each of n events has probability p, the probability that at least one event occurs is at most np

Height II

Consider a skip list with n entires

- By Fact 1, we insert an entry in list S_i with probability $1/2^i$
- By Fact 2, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3logn$, we have that the probability that S_{3logn} has at least one entry is at most $n/2^{3logn} = n/n^3 = 1/n^2$
- Thus a skip list with n entries has height at most $3logn$ with probability at least $1 - 1/n^2$

Search and Update Times

- The search time in a skip list is proportional to
  - the number of drop-down steps, plus
  - the number of scan-forward steps
- The drop-down steps are bounded by the height of the skip list and thus are $O(logn)$ with high probability
- To analyze the scan-forward steps, we use yet another probabilistic fact:
- **Fact 4**: The expected number of coin tosses required in order to get tails is 2

Search and Update Times

- When we scan forward in a list, the destination key does not belong to a higher list. A scan-forward step is associated with a former coin toss that gave tails
- By Fact 4, in each list the expected number of scan-forward steps is 2
- Thus, the expected number of scan-forward steps is $O(logn)$
- We conclude that a search in a skip list takes $O(logn)$ expected time
- The analysis of insertion and deletion gives similar results

Summary

- A skip list is a data structure for dictionaries that uses a randomized insertion algorithm
- In a skip list with n entries:
  - The expected space used is $O(n)$
  - The expected search, insertion and deletion time is $O(logn)$
- Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability
- Skip lists are fast and simple to implement in practice