Fall 2015

Fang Yu

Software Security Lab.
Dept. Management Information Systems,
National Chengchi University

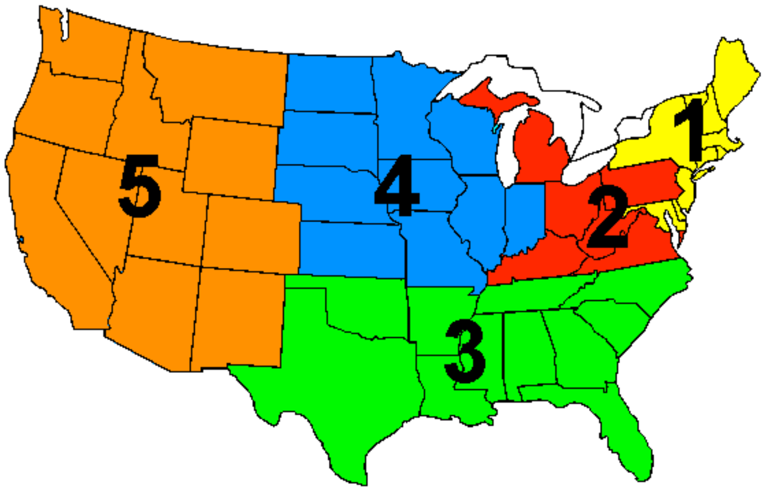
Data Structures Lecture 13 (reference)

Advance ADTs II

Ordered Maps, Dictionaries, Skip Lists, Sets, and Partitions

Ordered Maps

- Recall that a Map stores pairs of (key, value)
- In ordered maps, keys are assumed to come from a total order and pairs are stored in order
- New operations:
  - firstEntry(): entry with smallest key value null
  - lastEntry(): entry with largest key value
  - floorEntry(k):entry with largest key ≤ k
  - ceilingEntry(k): entry with smallest key ≥ k
  - These operations return null if the map is empty

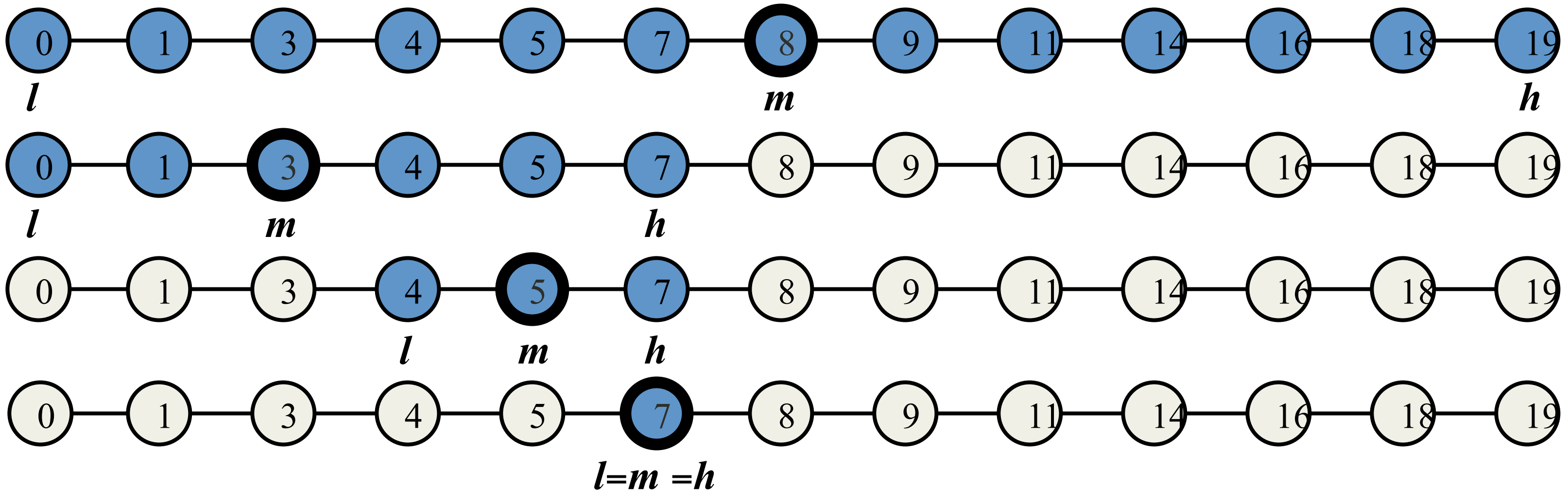


Binary Search

- Binary search can perform operations get, floorEntry and ceilingEntry on an ordered map 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 O(log n) steps

Binary Search

- Example: find(7)



Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
- We store the items in an array-based sequence, sorted by key
- We use an external comparator for the keys

Search Tables

- Performance:
  - get, floorEntry and ceilingEntry take $O(\log n)$ time, using binary search
  - insert takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
  - remove take $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal
- The lookup 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)

Dictionary

- A dictionary models a searchable collection of key-element 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 (e.g., datastructures.net) to internet IP addresses (e.g., 128.148.34.101)

Dictionary ADT

- Dictionary ADT methods:
  - get(k): if the dictionary has an entry with key k, returns it, else, returns null
  - getAll(k): returns an iterable collection of all entries with key k
  - put(k, o): inserts and returns the entry (k, o)
  - remove(e): remove the entry e from the dictionary
  - entrySet(): returns an iterable collection of the entries in the dictionary
  - size(), isEmpty()

Example

Operation	Output	Dictionary
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
get(4)	**null**	(5,A),(7,B),(2,C),(8,D),(2,E)
get(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
getAll(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(get(5))	(5,A)	(7,B),(2,C),(8,D),(2,E)
get(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:
  - put takes $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence
  - get 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
- 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)

The getAll and put Algorithms

```
Algorithm getAll(k)
Create an initially-empty list L
for e: D do
    if e.getKey() = k  then
            L.addLast(e)
return L

Algorithm put(k,v)
Create a new entry e = (k,v)
S.addLast(e)            {S is unordered}
return e
```

The remove Algorithm

```
Algorithm remove(e):
{ We don't assume here that e stores its position in S }
B = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element() = e then
            S.remove(p)
            return e
return null        {there is no entry e in D}
```

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.

Recall that

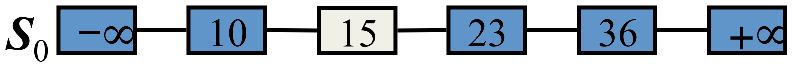
- A search table is a dictionary implemented by means of a sorted array
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - get takes $O(\log n)$ time, using binary search
  - put takes $O(n)$ time since in the worst case we have to shift n items to make room for the new item
  - remove takes $O(n)$ time since in the worst case we have to shift n items to compact the items after the removal

Recall that

- 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)

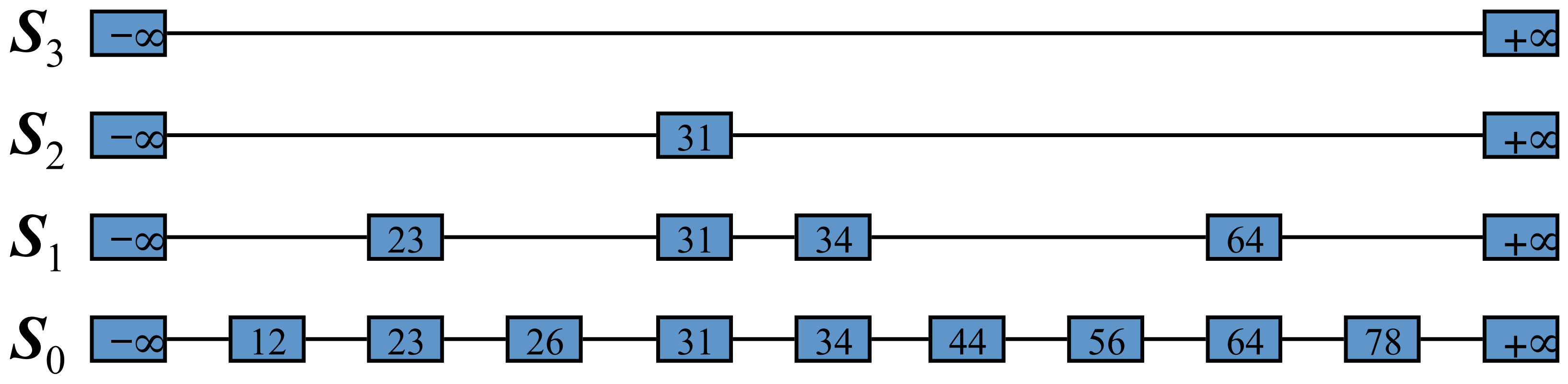
Skip Lists

- We can do better using skip lists
- Find the entry by jumping



What is a Skip List

- 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.,
    $$S_h \subseteq S_{h-1} \subseteq \ldots \subseteq S_0$$
  - List S_h contains only the two special keys



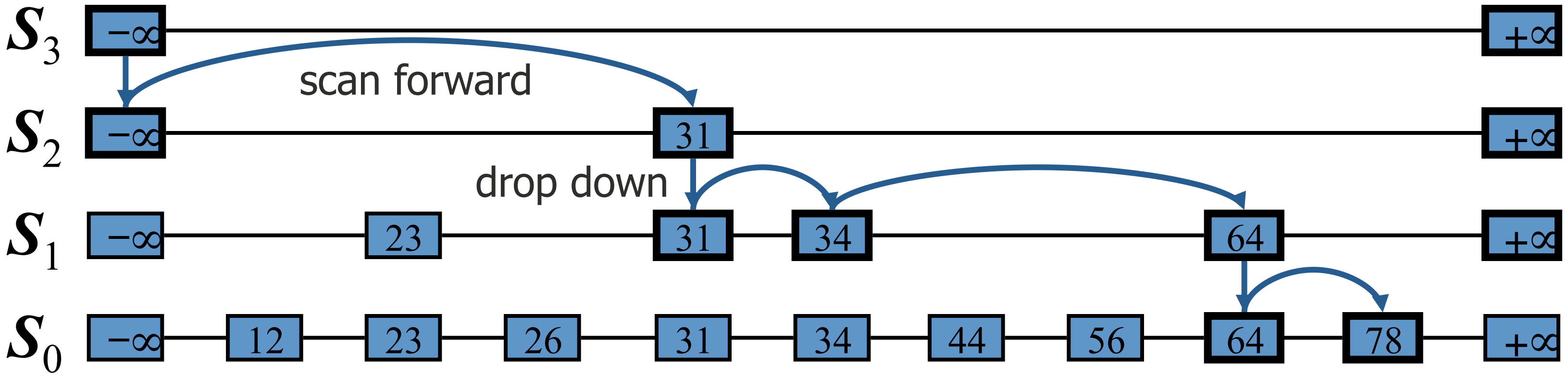
Search

- 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 $y \leftarrow key(next(p))$

    $x = y$: 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 $null$
- Example: search for 78



Randomized Algorithms

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

```
b ← random()
if  b = 0
        do A …
else { b = 1}
        do  B …
```

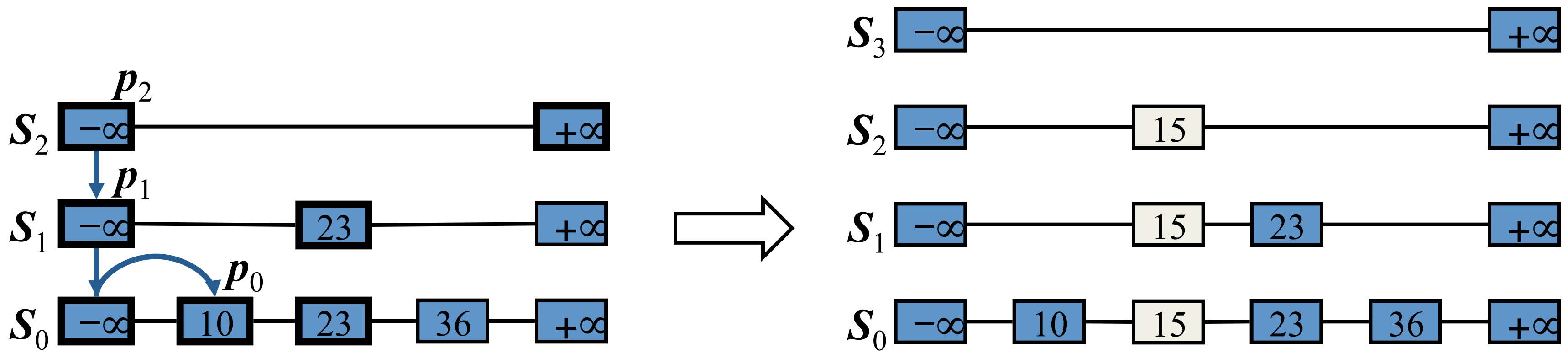
- Its running time depends on the outcomes of the coin tosses
- We analyze the expected running time of a randomized algorithm under the following assumptions
  - the coins are unbiased, and
  - the coin tosses are independent
- The worst-case running time of a randomized algorithm is often large but 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

Insertion

- To insert an entry (x, o) into a skip list:
  - 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 $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j

Insertion

- Example: insert key 15, with $i = 2$

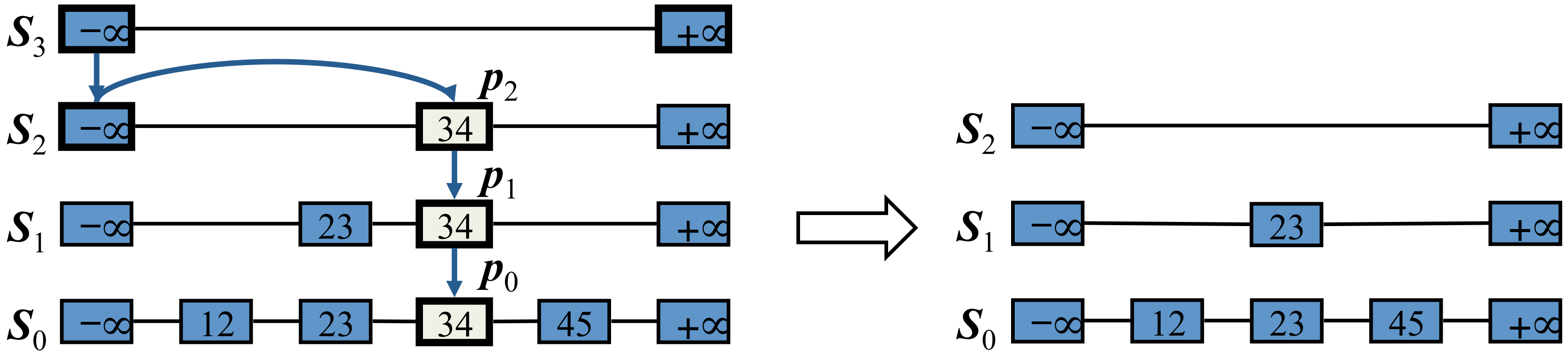


Deletion

- 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 $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys

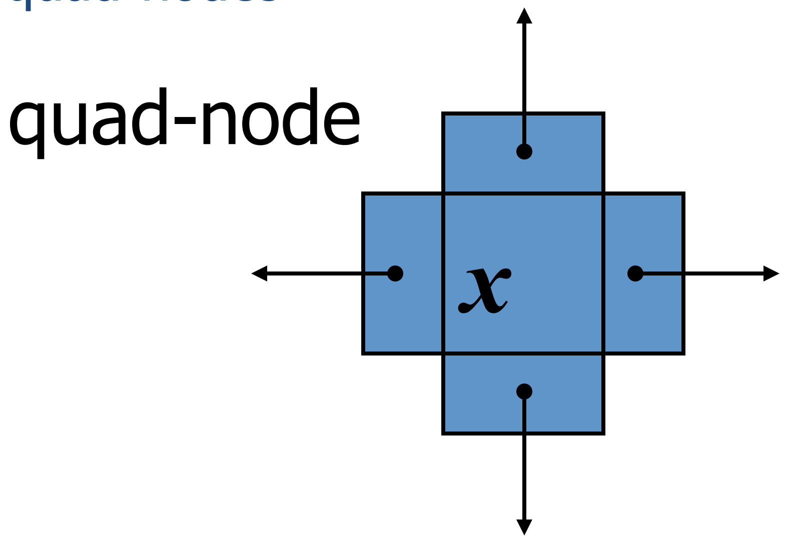
Deletion

- Example: remove key 34



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

- 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 $n/2^i$
- The expected number of nodes used by the skip list is

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

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

Height

- The running time of the search an 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 $O(\log n)$
- 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

- 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 3, the probability that list S_i has at least one item is at most $n/2^i$
- By picking $i = 3\log n$, we have that the probability that $S_{3\log n}$ has at least one entry is at most

  $$n/2^{3\log n} = n/n^3 = 1/n^2$$

- Thus a skip list with n entries has height at most $3\log n$ with probability at least $1 - 1/n^2$

Search and Update Times

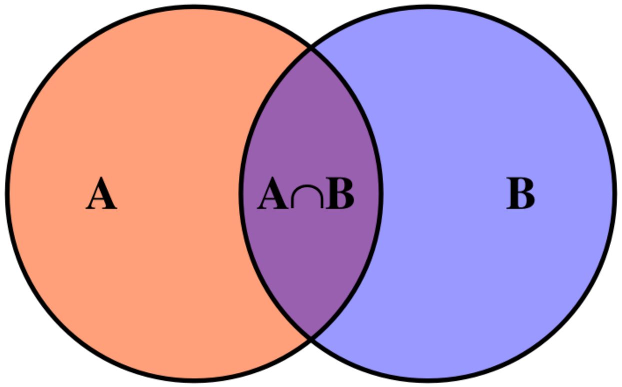
- The search time in a skip list is proportional to
  - the number of drop-down steps, plus
  - the number of scan-forward steps
- The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability
- To analyze the scan-forward steps, we use yet another probabilistic fact:
  The expected number of coin tosses required in order to get tails is 2
- 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 $O(\log n)$
- We conclude that a search in a skip list takes $O(\log n)$ expected time
- The analysis of insertion and deletion gives similar results

Summary

- A skip list is a data structure for ordered maps/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 $O(\log n)$
- 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

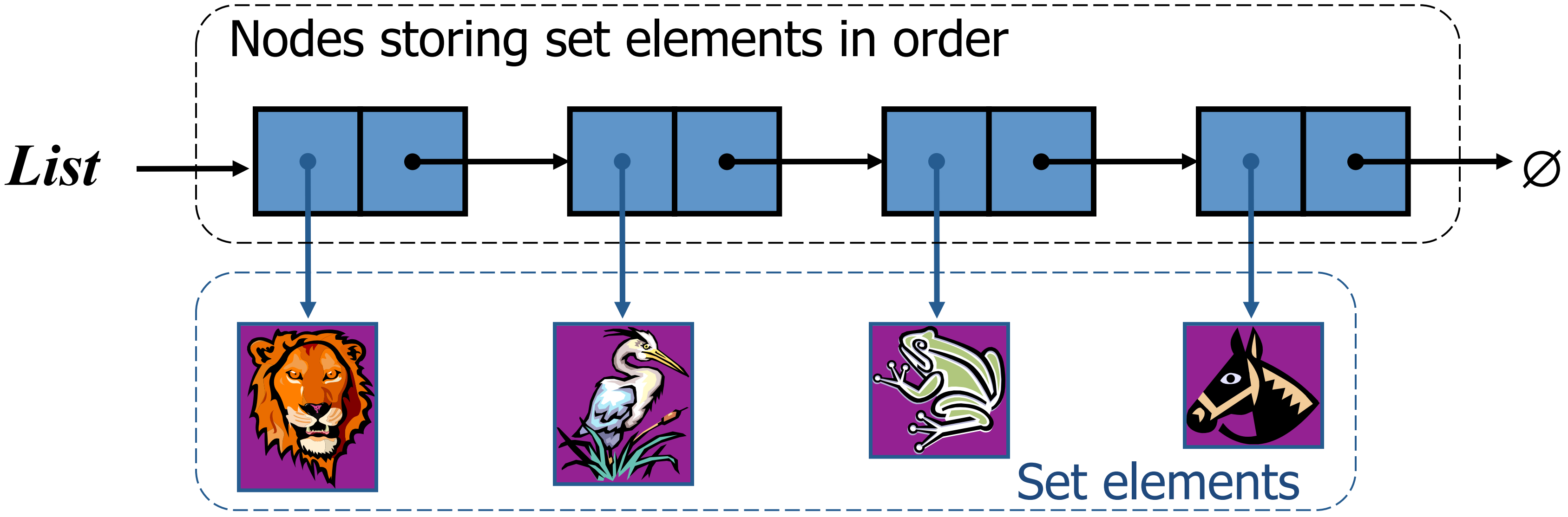
Sets

- A set is a collection of distinct objects
- The methods a the set ADT
  - Add(e)
  - Remove(e)
  - Contains(e)
  - Iterator()
- The operations between pairs of sets
  - A.union(B) : { x | x is in A or x is in B }
  - A.intersect(B) : { x | x is in A and x is in B }
  - A.subtract(B) : { x | x is in A and x is not in B }



Storing a Set in a List

- We can implement a set with a list
- Elements are stored sorted according to some canonical ordering
- The space used is $O(n)$



Generic Merging

- Generalized merge of two sorted lists A and B
- Template method genericMerge
- Auxiliary methods
  - aIsLess
  - bIsLess
  - bothAreEqual
- Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        a ← A.first().element();  b ← B.first().element()
        if a < b
            aIsLess(a, S);  A.remove(A.first())
        else if b < a
            bIsLess(b, S);  B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first());  B.remove(B.first())
    while ¬A.isEmpty()
        aIsLess(a, S);  A.remove(A.first())
    while ¬B.isEmpty()
        bIsLess(b, S);  B.remove(B.first())
    return S
```

Using Generic Merge for Set Operations

- Any of the set operations can be implemented using a generic merge
- For example:
  - For intersection: only copy elements that are duplicated in both list
  - For union: copy every element from both lists except for the duplicates
- All methods run in linear time

- Set union:
  - *aIsLess*(*a*, *S*)

    S.insertLast(*a*)
  - *bIsLess*(*b*, *S*)

    S.insertLast(*b*)
  - *bothAreEqual*(*a*, *b*, *S*)

    S. insertLast(*a*)
- Set intersection:
  - *aIsLess*(*a*, *S*)

    { *do nothing* }
  - *bIsLess*(*b*, *S*)

    { *do nothing* }
  - *bothAreEqual*(*a*, *b*, *S*)

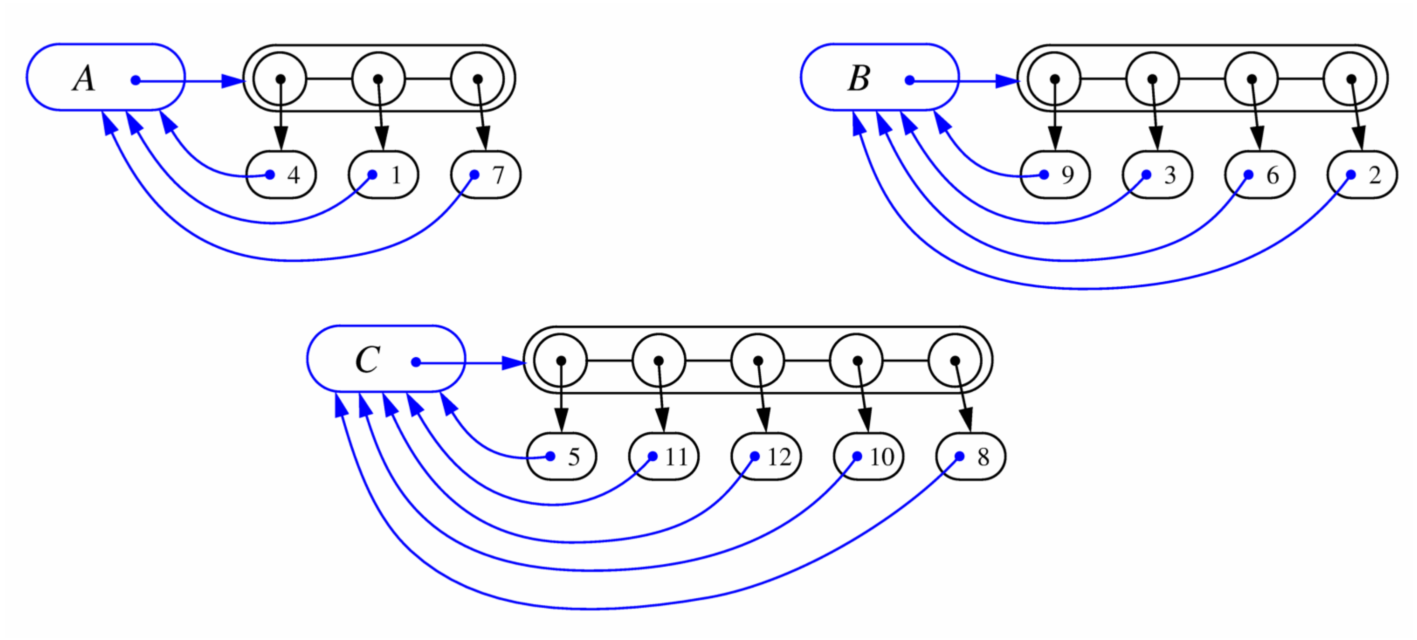
    S. insertLast(*a*)

Partitions

- A partition is a collection of disjoint sets
  - E.g., A = {1, 4, 7}, B= {2, 3, 6, 9}, C = {5, 8, 10, 11, 12}
- The methods of the partition ADT
  - makeSet(x): Create a singleton set containing the element x and return the position storing x in this set
  - union(A,B): Return the set A U B, destroying the old A and B
  - find(p): Return the set containing the element at position p

List-based Implementation

- Each set is stored in a sequence represented with a linked-list
- Each node should store an object containing the element and a reference to the set name

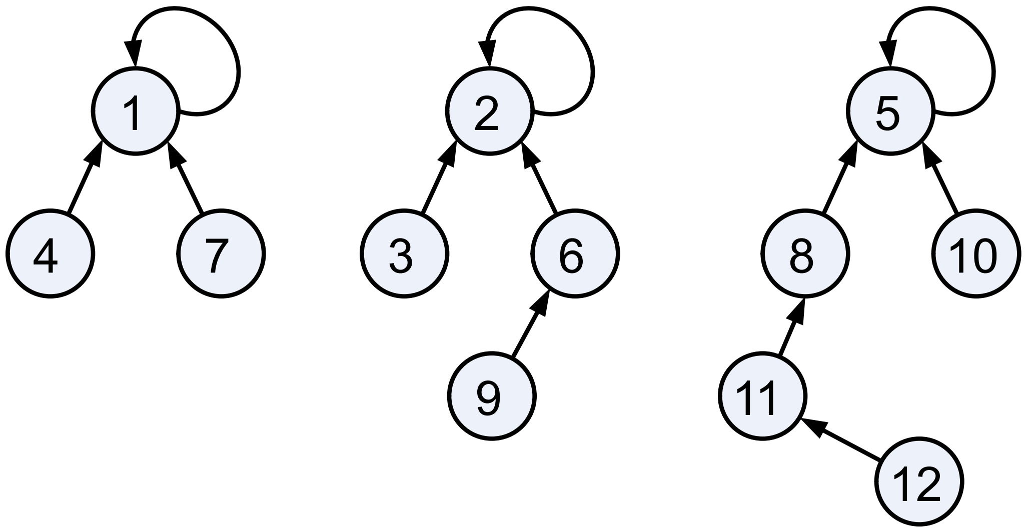


Analysis of List-based Representation

- When doing a union, always move elements from the smaller set to the larger set
  - Each time an element is moved it goes to a set of size at least double its old set
  - Thus, an element can be moved at most O(log n) times
- Total time needed to do n unions and finds is O(n log n).

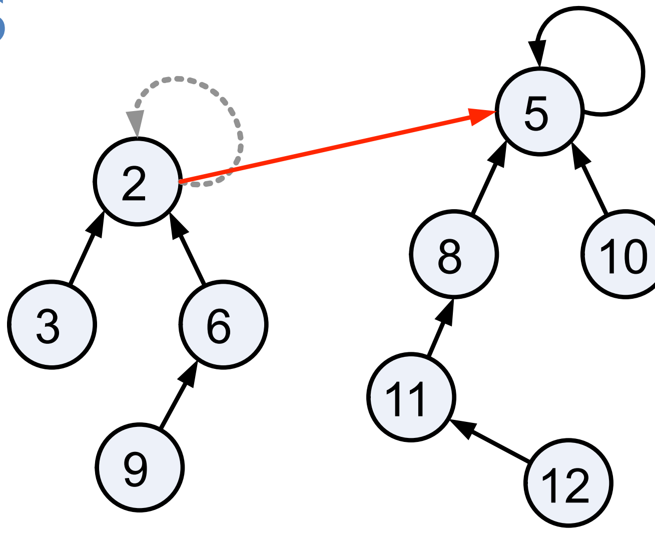
Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a set name
- A node v whose set pointer points back to v is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- For example: The sets “1”, “2”, and “5”:

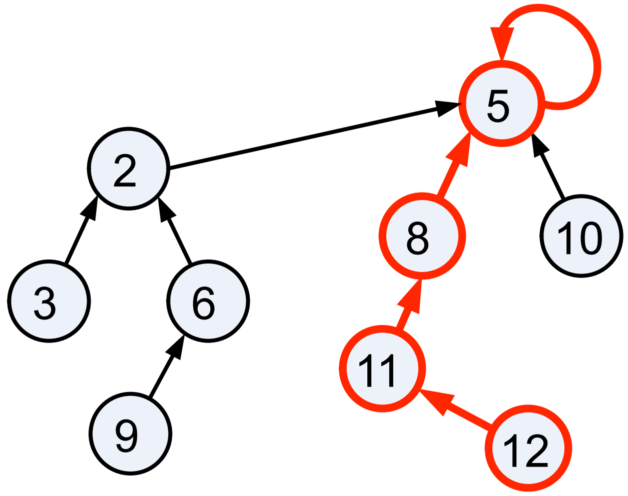


Union-Find Operations

- To do a union, simply make the root of one tree point to the root of the other

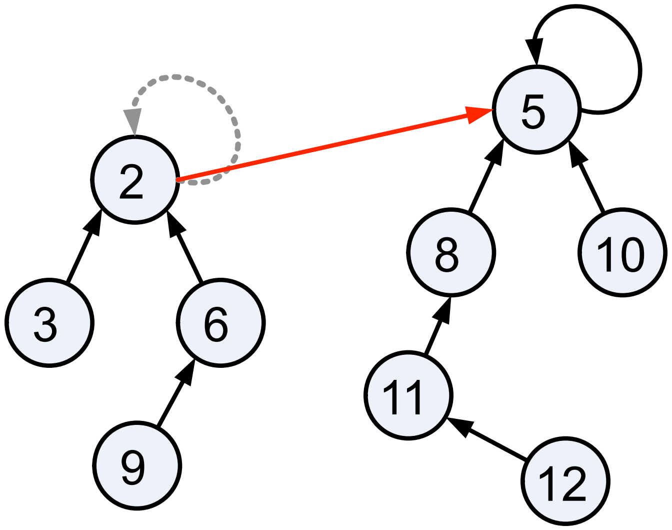


- To do a find, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself



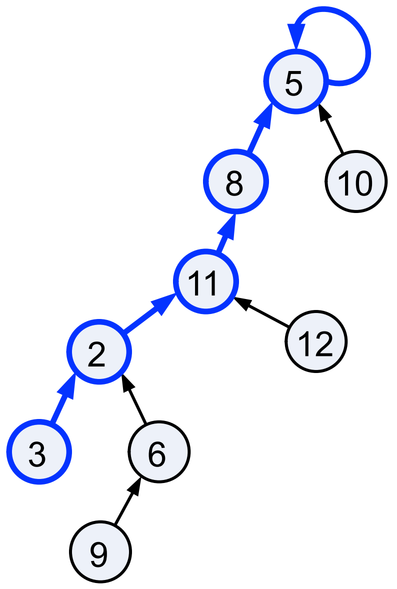
Union-Find Heuristic 1

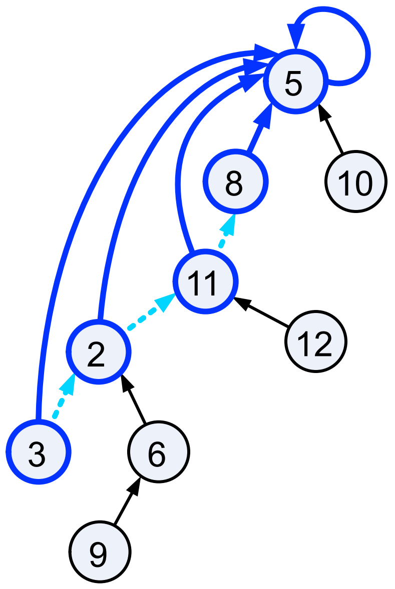
- Union by size:
  - When performing a union, make the root of smaller tree point to the root of the larger
- Implies O(n log n) time for performing n union-find operations:
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most O(log n) pointers for any find.



Union-Find Heuristic 2

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root





- Implies O(n log* n) time for performing n union-find operations:

Bonus HW12 (Due on Jan. 8)

Classify webpages by keywords!

- Given a set of web pages and a set of keywords, classify webpages by whether it contains the keyword
- For each keyword k, store a set, W(k), of Web pages that contain keyword k
- You can implement the set using an ordered linked list (in an alphabetical order)
- Implement set operations using genericMerge algorithm:
  - Intersection, Union, Subtraction
- You can also use the java library that implements java.util.Set, e.g., java.util.HashSet
- You can deal with two-word query, i.e., webpages that contain both k1 and k2, by returning W(k1) intersect W(k2)

Operations

Given a sequence of operations in a txt file, parse the txt file and execute each operation accordingly

operations	description
Add(set k, webpages)	Add webpages that contain a keyword k to the set k
Union(set k1, set k2)	Return all webpages that contain k1 or k2
Intersect(set k1, set k2)	Return all webpages that contain k1 and k2
Subtract(set k1, set k2)	Return all webpages that contain k1 but do not contain k2

An input file

Similar to HW12,

1. You need to read the sequence of operations from a txt file
2. The format is firm
3. Raise an exception if the input does not match the format

```
AddKey Fang
AddKey Dissertation ICSE Yu
AddWeb www.cs.ucsb.edu/~yuf
AddWeb www3.nccu.edu.tw/~yuf
Intersect Fang ICSE
Union Fang Dissertation
Subtract Fang Yu
```

Output:

```
Web Pages that contains Fang and ICSE:
www3.nccu.edu.tw/~yuf

Web Pages that contains Fang or Dissertation:
www.cs.edu.tw/~yuf
www3.nccu.edu.tw/~yuf

Web Pages that contains Fang but does not
contain Yu:
None in the target webpages
```