COMP9024: Data Structures and Algorithms

Randomized Algorithms

Contents

Randomized Algorithm (1/2)

- A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic.
  - The algorithm typically uses uniformly random bits as an auxiliary input to guide its behaviour, in the hope of achieving good performance in the average case over all possible choices of random bits.
- The performance of a randomized algorithm is a random variable determined by the random bits.
  - The worst-case performance is typically bad with a very small probability but the average performance can be good.
- Two categories: Las Vegas algorithm and Monte Carlo algorithm

Randomized Algorithm (2/2)

- Las Vegas algorithm
  - A Las Vegas algorithm is a randomized algorithm that always gives correct results
- Monte Carlo algorithm
  - A Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a certain (typically small) probability

An Example (1/5)

- Given an unsorted list where half of the elements have a key k1 and the other half have a key k2, find an element in the list with key k1.

An Example (2/5)

- Las Vegas algorithm

```
Algorithm findKey(L, k1)
Input: list L, key k1
Output: an element in L with key k1
{
   repeat
      randomly select e∈L;
   until key(e)=k1;
   return e;
}
```

An Example (3/5)

Analysis:

- Probability of success: 1
- The number of iterations varies and can be arbitrarily large, but the expected number of iterations is:

$$\lim_{n\to\infty} \sum_{i=0}^{n} \frac{i}{2^i} = 2$$

- The expected time complexity is O(1)

An Example (4/5)

- Monte Carlo algorithm

```
Algorithm findKey(L, k1)
Input: list L, key k1
Output: an element in L with key k1
{
   i=0;
   repeat
       randomly select e∈L;
       i++;
   until key(e)=k1 or i=m;
   return e;
}
```

An Example (5/5)

Analysis:

- After k iterations, the probability of finding an element with key k1 is $1 - \left(\frac{1}{2}\right)^k$
- Time complexity is O(1) but its does not guarantee success

Quick Selection

The Selection Problem

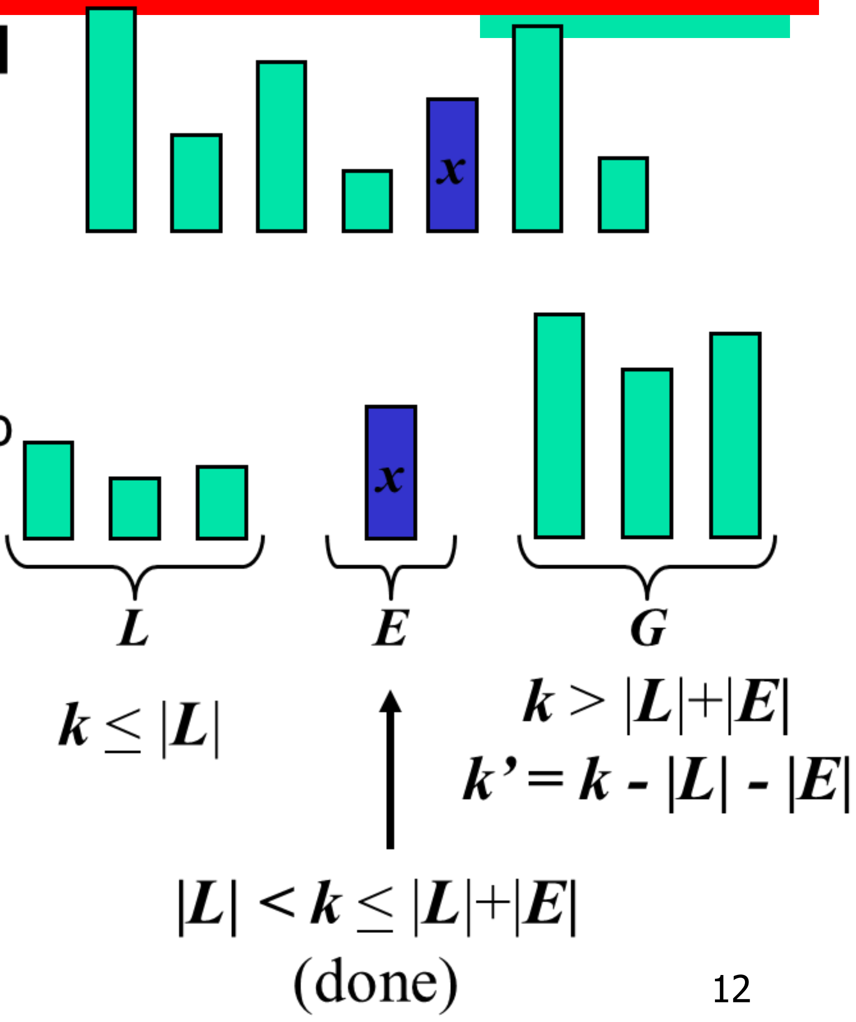
- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the k-th smallest element in this set.
- Of course, we can sort the set in O(n log n) time and then index the k-th element.

k=3 $\boxed{7\ \ 4\ \ 9\ \ \underline{6}\ \ 2 \rightarrow 2\ \ 4\ \ \underline{6}\ \ 7\ \ 9}$

- Can we solve the selection problem faster?

Quick-Select (1/2)

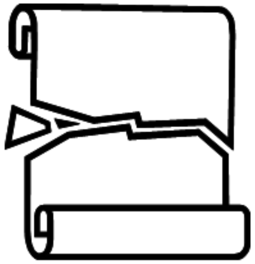
- Quick-select is a **randomized** selection algorithm based on the prune-and-search paradigm:
  - Prune: pick a random element x (called pivot) and partition S into
    - L elements less than x
    - E elements equal x
    - G elements greater than x
  - Search: depending on k, either answer is in E, or we need to recurse in either L or G



Quick-Select (2/2)

```
Algorithm quickSelect(S, k)
  Input Sequence S of |S| comparable elements and an integer k in [1, |S|]
  Output The k-th smallest element of S
  { if |S|=1
      return the (first) element of S;
    pick a random integer i in [1, |S|]; // |S| is the size of S;
    (L, E, G)=partition(S, i);
    if  k < l
       quickSelect(L, k);
    else if  k <= |L|+|E|
            return S[i]; // Each element in E is equal to S[i]
         else //  Find the k-|L|-|E|-th element in G
            quickSelect(G, k-|L|-|E|);
  }
```

Partition

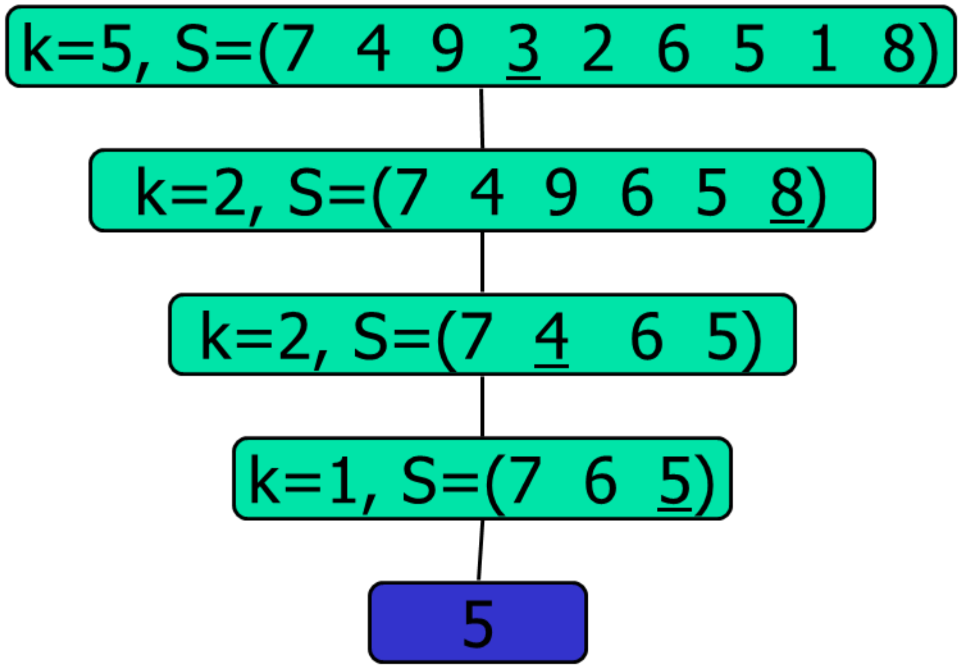


- We partition an input sequence as in the quick-sort algorithm:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-select takes $O(n)$ time

```
Algorithm partition($S, i$)
    Input sequence $S$, index $i$ of the pivot
    Output subsequences $L, E, G$ of the
    elements of $S$ less than, equal to,
    or greater than the pivot, resp.
  { $L, E, G$ = empty sequences;
    $x = S[i]$;
   while ( $\neg S.isEmpty()$ )
     { $y = S.remove(S.first())$;
       if ( $y < x$ )
          $L.insertLast(y)$;
       else if ( $y = x$ )
          $E.insertLast(y)$;
       else // $y > x$
          $G.insertLast(y)$; }
    return $L, E, G$;
  }
```

Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



Worst-Case Time complexity (1/3)

- What is the worst-case?
  - Each time the smallest key or the largest key is selected, and thus only one element (the one with the smallest key or the one with the largest key) is excluded on each iteration
- The worst-case time complexity is
- $O(n)+O(n-1)+\ldots+O(2)+O(1)$

$=O((n(n+1)/2))=O(n^2)$

Worst-Case Time complexity (2/3)

- What is the probability of the worst-case?
- Let $Ɛ_k$ be the event that we pick the largest or smallest element when there are k elements left.
- Let event Ɛ be the worst-case. We have:

$$Ɛ=\prod_{i=1}^{i=n} \varepsilon_i$$

- What is $P(Ɛ)=P(\prod_{i=1}^{i=n} \varepsilon_i)$?
- Since all $Ɛ_i$'s are independent, this simplifies to

$$P(Ɛ)=P(\prod_{i=1}^{i=n} \varepsilon_i)=\prod_{i=1}^{i=n} P(\varepsilon_i)$$

Worst-Case Time complexity (3/3)

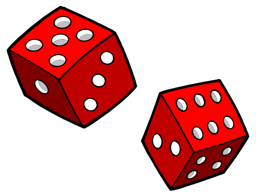
- $P(\mathcal{E}_1) = 1$.
- If $i>1$, then $P(\mathcal{E}_i)=\frac{2}{i}$. Thus

$$P(\mathcal{E})=\prod_{i=1}^{i=n} P(\varepsilon_i)=\prod_{i=2}^{i=n} \frac{2}{i} = \frac{2^{n-1}}{n!}$$

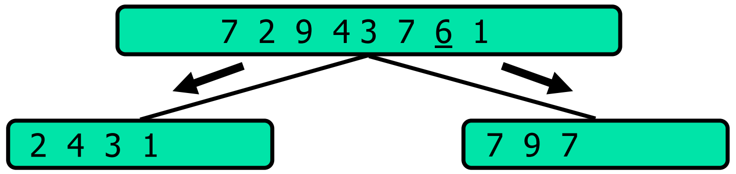
- if $n = 31$, then $2^{n-1}<10^{10}$ and $n! \approx 8 \times 10^{33}$

  $P(\mathcal{E})<1/10^{22}$. This is extremely unlikely!

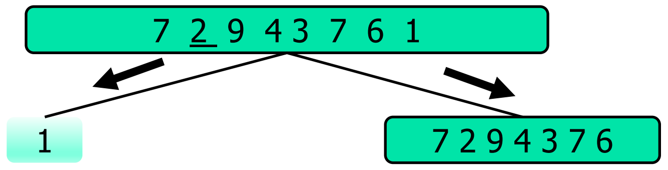
Expected Running Time (1/2)



- Consider a recursive call of quick-select on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

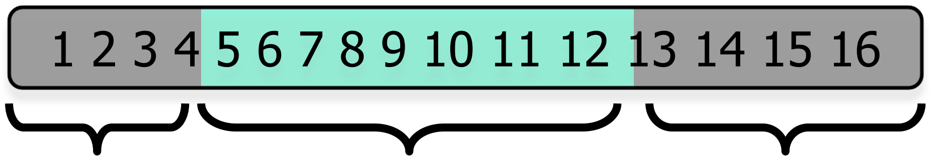


Good call



Bad call

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:

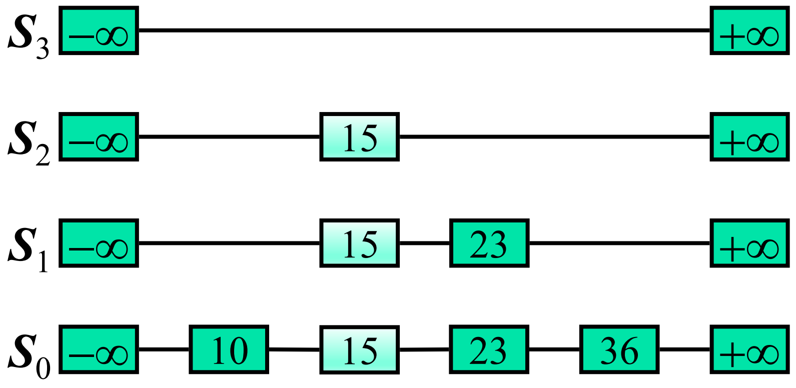


Bad pivots **Good pivots** **Bad pivots**

Expected Running Time (2/2)

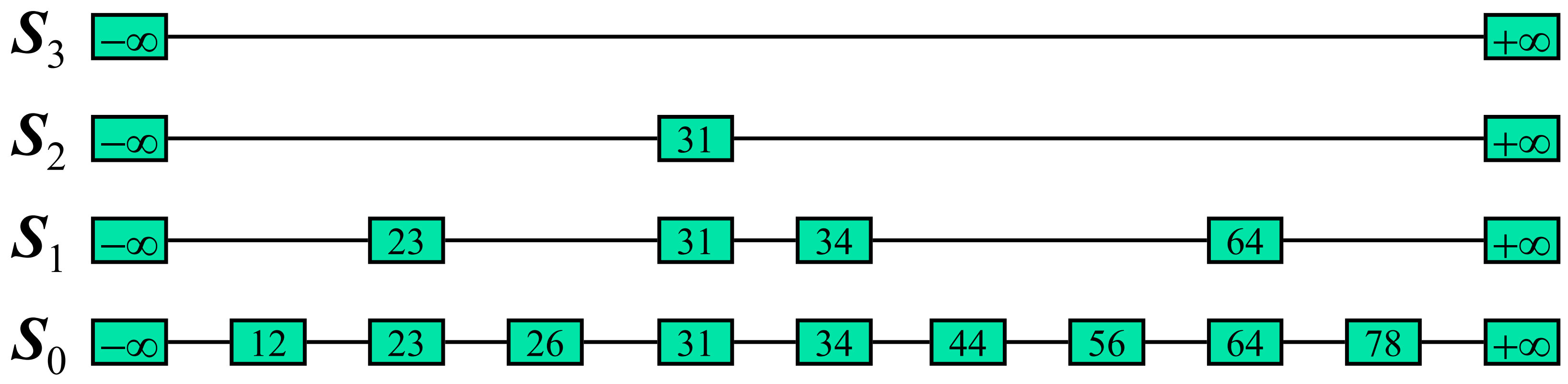
- Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two
- Probabilistic Fact #2: Expectation is a linear function:
  - $E(X+Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$
- Let T(n) denote the expected running time of quick-select.
- By Fact #2,
  - $T(n) \le T(3n/4) + bn$*(expected # of calls for a good call)
- By Fact #1,
  - $T(n) \le T(3n/4) + 2bn$
- That is, T(n) is a geometric series:
  - $T(n) \le 2bn + 2b(3/4)n + 2b(3/4)^2 n + 2b(3/4)^3 n + \ldots$
- So T(n) is O(n).
- We can solve the selection problem in O(n) expected time.

Skip Lists


S_3
$-\infty$
$+\infty$
S_2
$-\infty$
15
$+\infty$
S_1
$-\infty$
15
23
$+\infty$
S_0
$-\infty$
10
15
23
36
$+\infty$

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.,
    $$S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$$
  - List S_h contains only the two special keys
- We show how to use a skip list to implement the dictionary ADT



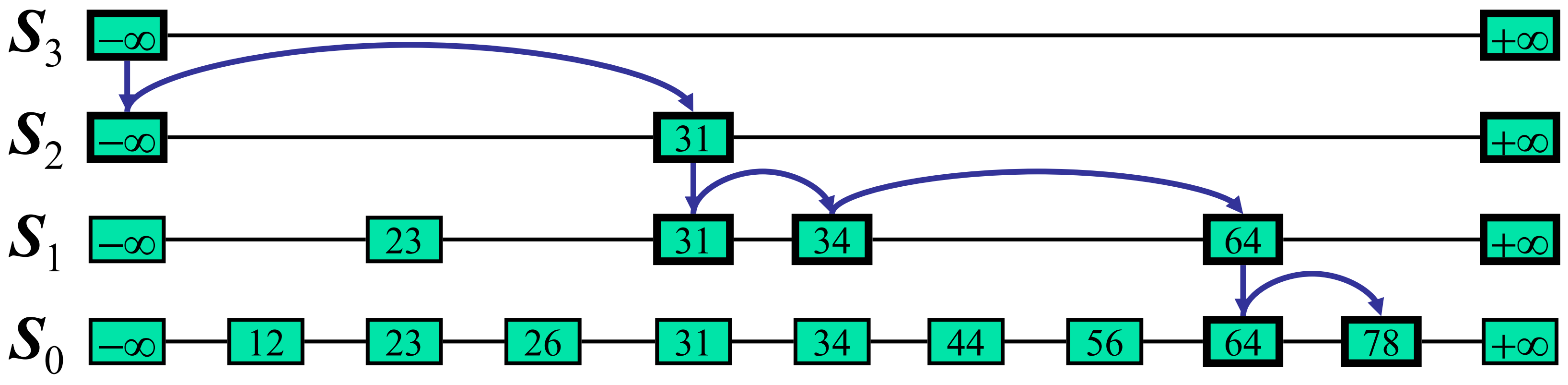
Search

- 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 $element(next(p))$

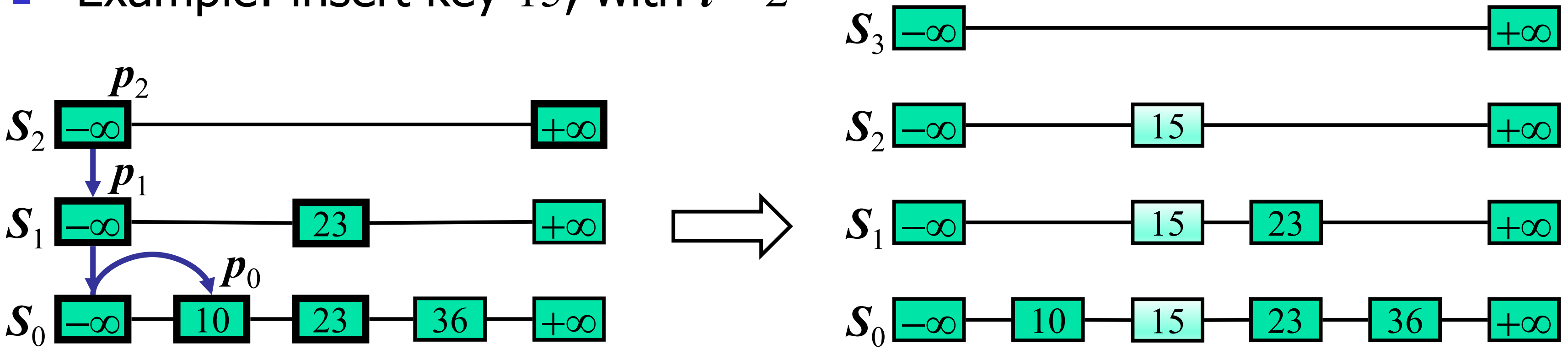
    $x > y$: we "scan forward"

    $x < y$: we "drop down"
  - If we try to drop down past the bottom list, we return ***null***
- Example: search for 78



Insertion

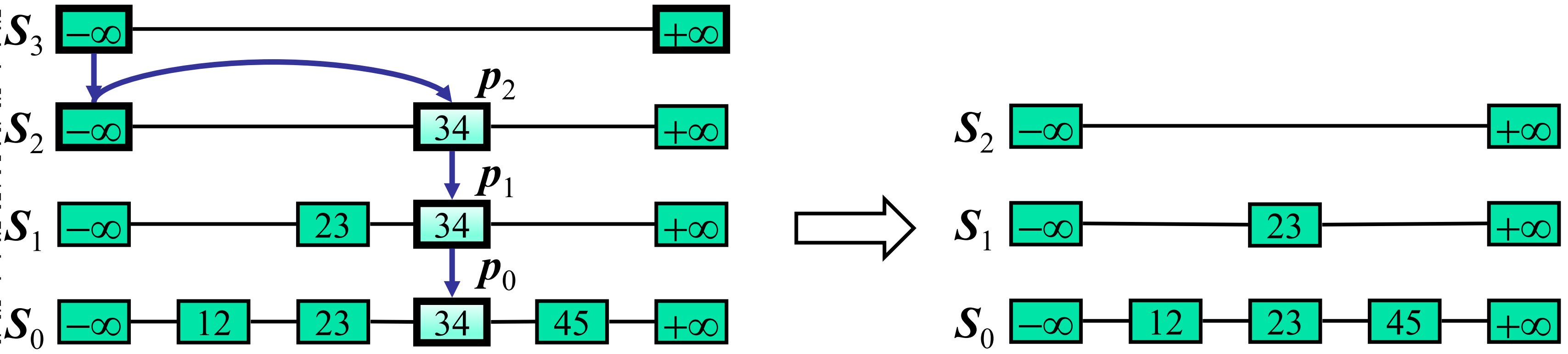
- To insert an entry (x, o) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get a tail, and we denote with i the number of times the coin came up with heads
  - If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the 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
- Example: insert key 15, with $i = 2$



p_2
S_2 $-\infty$ $+\infty$
p_1
S_1 $-\infty$ 23 $+\infty$
p_0
S_0 $-\infty$ 10 23 36 $+\infty$
S_3 $-\infty$ $+\infty$
S_2 $-\infty$ 15 $+\infty$
S_1 $-\infty$ 15 23 $+\infty$
S_0 $-\infty$ 10 15 23 36 $+\infty$

Deletion

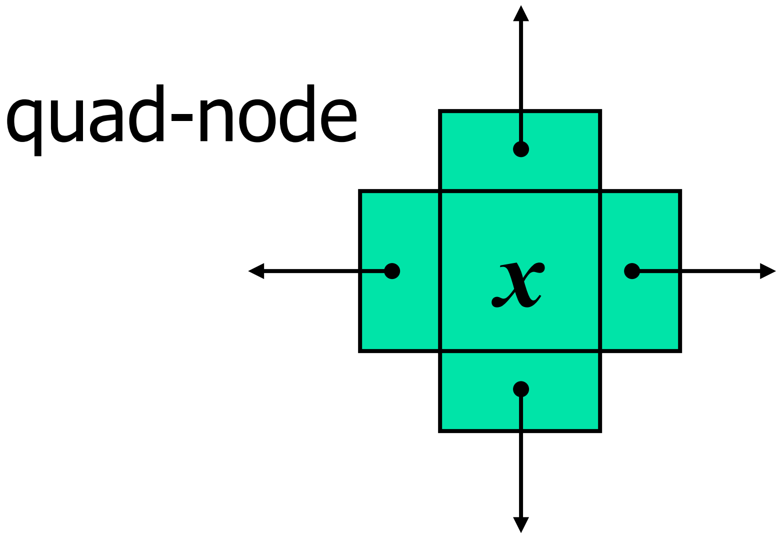
- To remove an entry with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_j is in list S_j
  - We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
  - We remove all but one list containing only the two special keys
- Example: remove key 34



S_3 $-\infty$ $+\infty$
S_2 $-\infty$ 34 $+\infty$
p_2
p_1
p_0
S_1 $-\infty$ 23 34 $+\infty$
S_0 $-\infty$ 12 23 34 45 $+\infty$
S_2 $-\infty$ $+\infty$
S_1 $-\infty$ 23 $+\infty$
S_0 $-\infty$ 12 23 45 $+\infty$

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 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 $O(\log n)$
- 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:

  Fact 4: The expected number of coin tosses required in order to get a tail 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 a tail
- 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

- Randomized algorithm
- Las Vegas algorithm
- Monte Carlo algorithm
- Randomized selection algorithm
- Skip lists