Skip lists and other search structures

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

18 November 2013

Introduction

- One of the most basic tasks in computer science is searching for data.

Introduction

- One of the most basic tasks in computer science is searching for data.
- In this task's most basic form, we imagine that we have a "database" containing a number of records, each consisting of a key, and some associated data.

Introduction

- One of the most basic tasks in computer science is searching for data.
- In this task's most basic form, we imagine that we have a "database" containing a number of records, each consisting of a key, and some associated data.
- We can search for keys, and want to find the associated data.

Introduction

- One of the most basic tasks in computer science is searching for data.
- In this task's most basic form, we imagine that we have a "database" containing a number of records, each consisting of a key, and some associated data.
- We can search for keys, and want to find the associated data.
- For example, the database might be a list of students: the key might be a student ID, and the data might be everything else associated with that student (name, address, etc.).

Introduction

- One of the most basic tasks in computer science is searching for data.
- In this task's most basic form, we imagine that we have a "database" containing a number of records, each consisting of a key, and some associated data.
- We can search for keys, and want to find the associated data.
- For example, the database might be a list of students: the key might be a student ID, and the data might be everything else associated with that student (name, address, etc.).
- "Database" is in quotes because it is a database in the abstract sense, rather than (necessarily) a real-world database like MySQL. . .

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database
- Delete(k): Deletes the record with key k from the database

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database
- Delete(k): Deletes the record with key k from the database
- Find(k): Returns the data corresponding to the record whose key is k, or "not found"

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database
- Delete(k): Deletes the record with key k from the database
- Find(k): Returns the data corresponding to the record whose key is k, or "not found"
- Successor(k): Returns the key which is next in the database after k. Used to, for example, print a list of all records in order.

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database
- Delete(k): Deletes the record with key k from the database
- Find(k): Returns the data corresponding to the record whose key is k, or "not found"
- Successor(k): Returns the key which is next in the database after k. Used to, for example, print a list of all records in order.

There are a number of different data structures we could use for this task, with varying complexities.

- To compare them, we consider a simple version of the search problem where each key is an integer between 1 and U (the universe size).

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database
- Delete(k): Deletes the record with key k from the database
- Find(k): Returns the data corresponding to the record whose key is k, or "not found"
- Successor(k): Returns the key which is next in the database after k. Used to, for example, print a list of all records in order.

There are a number of different data structures we could use for this task, with varying complexities.

- To compare them, we consider a simple version of the search problem where each key is an integer between 1 and U (the universe size).
- We assume that the largest number of records ever stored in the database is n.

Introduction

We would like to support the following operations.

- Insert(k, d): Inserts a record with key k and data d into the database
- Delete(k): Deletes the record with key k from the database
- Find(k): Returns the data corresponding to the record whose key is k, or "not found"
- Successor(k): Returns the key which is next in the database after k. Used to, for example, print a list of all records in order.

There are a number of different data structures we could use for this task, with varying complexities.

- To compare them, we consider a simple version of the search problem where each key is an integer between 1 and U (the universe size).
- We assume that the largest number of records ever stored in the database is n.
- In general, we imagine that n is much smaller than U.

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

Insert(7, Alice)

Ashley Montanaro
ashley@cs.bris.ac.uk

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

Insert(3, Bob)

Ashley Montanaro
ashley@cs.bris.ac.uk

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

Successor(3)

Ashley Montanaro
ashley@cs.bris.ac.uk

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

returns 7

Ashley Montanaro
ashley@cs.bris.ac.uk

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

- Insert, Delete and Find are all very efficient ($\Theta(1)$) but Successor could take time $\Theta(U)$, as we need to seach through all subsequent elements in the array in turn.

Ashley Montanaro
ashley@cs.bris.ac.uk

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

		Bob				Alice			

- Insert, Delete and Find are all very efficient ($\Theta(1)$) but Successor could take time $\Theta(U)$, as we need to seach through all subsequent elements in the array in turn.
- Perhaps more importantly, the array uses $\Theta(U)$ space, even if it only contains a small number of elements!

Array

- The simplest data structure we could use would be an array.
- Here we simply store the record with key k at position k in the array.

		Bob				Alice			

- Insert, Delete and Find are all very efficient ($\Theta(1)$) but Successor could take time $\Theta(U)$, as we need to seach through all subsequent elements in the array in turn.
- Perhaps more importantly, the array uses $\Theta(U)$ space, even if it only contains a small number of elements!

Structure	Space used	Insert	Delete	Find	Successor
Array	$\Theta(U)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(U)$

Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.

Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.

Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.
- This allows complexities to be obtained that do not depend on U.

$\boxed{\text{list head}}$

Unsorted linked list

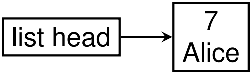
- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.
- This allows complexities to be obtained that do not depend on U.

Insert(7, Alice) $\boxed{\text{list head}}$

Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.
- This allows complexities to be obtained that do not depend on U.

Insert(3, Bob)

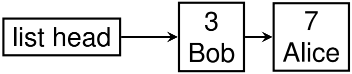


Ashley Montanaro
ashley@cs.bris.ac.uk

Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.
- This allows complexities to be obtained that do not depend on U.

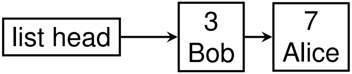
Successor(3)



Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.
- This allows complexities to be obtained that do not depend on U.

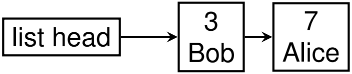
returns 7



Unsorted linked list

- We could also consider storing the records in unsorted order in a linked list.
- When a new record comes in, we just insert it at the start of the list.
- This allows complexities to be obtained that do not depend on U.

returns 7



This also makes Successor much quicker, but search and deletion slower:

Structure	Space used	Insert	Delete	Find	Successor
Unsorted linked list	$O(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(n)$

Exercise: Does it help to store the records sorted by key?

Ashley Montanaro
ashley@cs.bris.ac.uk

University of BRISTOL

Hash table

- Another data structure you have seen is the hash table.

Hash table

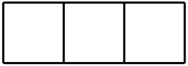
- Another data structure you have seen is the hash table.
- We store the table in an array of size m, where m is much smaller than U.

Hash table

- Another data structure you have seen is the hash table.
- We store the table in an array of size m, where m is much smaller than U.
- Given a key k, we compute a function $h(k)$ giving the position of k in the array.

Hash table

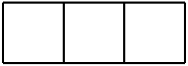
- Another data structure you have seen is the hash table.
- We store the table in an array of size m, where m is much smaller than U.
- Given a key k, we compute a function $h(k)$ giving the position of k in the array.
- If $m < U$, there must exist records that hash to the same position. To deal with this situation, we have a linked list at each element of the array to store these elements.



Hash table

- Another data structure you have seen is the hash table.
- We store the table in an array of size m, where m is much smaller than U.
- Given a key k, we compute a function $h(k)$ giving the position of k in the array.
- If $m < U$, there must exist records that hash to the same position. To deal with this situation, we have a linked list at each element of the array to store these elements.

Insert(7, Alice)

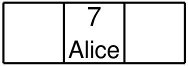


Ashley Montanaro
ashley@cs.bris.ac.uk

Hash table

- Another data structure you have seen is the hash table.
- We store the table in an array of size m, where m is much smaller than U.
- Given a key k, we compute a function $h(k)$ giving the position of k in the array.
- If $m < U$, there must exist records that hash to the same position. To deal with this situation, we have a linked list at each element of the array to store these elements.

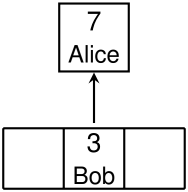
Insert(3, Bob)



Ashley Montanaro
ashley@cs.bris.ac.uk

Hash table

- Another data structure you have seen is the hash table.
- We store the table in an array of size m, where m is much smaller than U.
- Given a key k, we compute a function $h(k)$ giving the position of k in the array.
- If $m < U$, there must exist records that hash to the same position. To deal with this situation, we have a linked list at each element of the array to store these elements.



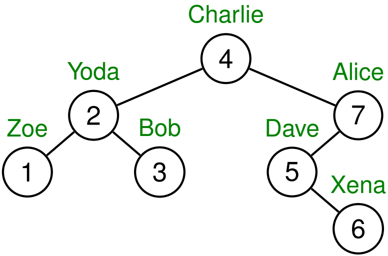
Hash table

In the worst case, we might have n records coming in which all hash to the same position. Then the complexity is no better than an unsorted linked list!

Structure	Space used	Insert	Delete	Find	Successor
Hash table	$O(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(n)$

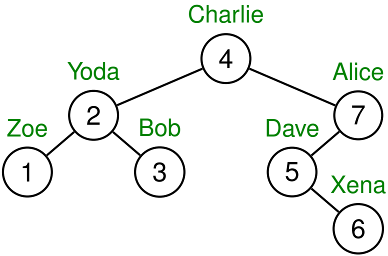
Binary tree

A binary tree offers another way to store data, and to list it easily.



Binary tree

A binary tree offers another way to store data, and to list it easily.

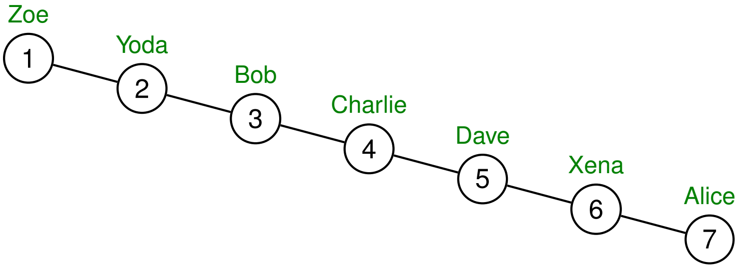


Now the complexities of the various operations all depend on the height h of the tree.

Structure	Space used	Insert	Delete	Find	Successor
Binary tree	$O(n)$	$O(h)$	$O(h)$	$O(h)$	$O(h)$

Binary tree

If the height is large, these operations are all inefficient. This can happen if keys are inserted into the tree in an unfortunate (e.g. ascending) order.

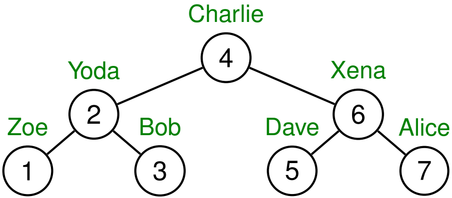


Structure	Space used	Insert	Delete	Find	Successor
Binary tree (worst case)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

In the worst case, this is even worse than an unsorted linked list.

AVL tree

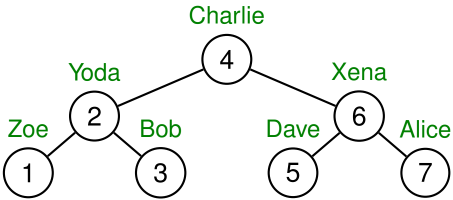
As you heard in COMS11600, an AVL tree is a binary tree which is maintained such that the height of a tree containing n keys is $O(\log n)$.



Structure	Space used	Insert	Delete	Find	Successor
AVL tree	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

AVL tree

As you heard in COMS11600, an AVL tree is a binary tree which is maintained such that the height of a tree containing n keys is $O(\log n)$.

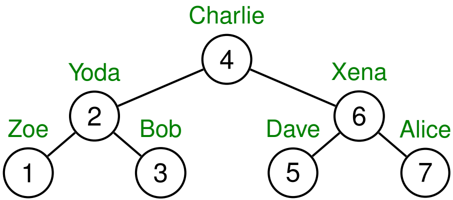


Structure	Space used	Insert	Delete	Find	Successor
AVL tree	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

- We have now achieved much better complexities, but at the expense of having a more complicated data structure.

AVL tree

As you heard in COMS11600, an AVL tree is a binary tree which is maintained such that the height of a tree containing n keys is $O(\log n)$.



Structure	Space used	Insert	Delete	Find	Successor
AVL tree	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

- We have now achieved much better complexities, but at the expense of having a more complicated data structure.
- Can we do better?

Skip lists

- We will discuss a way in which we can get complexities which match the above AVL tree bounds, with a much simpler data structure, called the skip list.

Skip lists

- We will discuss a way in which we can get complexities which match the above AVL tree bounds, with a much simpler data structure, called the skip list.
- The snag is that the complexities we obtain will be randomised.

Skip lists

- We will discuss a way in which we can get complexities which match the above AVL tree bounds, with a much simpler data structure, called the skip list.
- The snag is that the complexities we obtain will be randomised.
- That is, when we build our data structure, we will do it by tossing coins.

Skip lists

- We will discuss a way in which we can get complexities which match the above AVL tree bounds, with a much simpler data structure, called the skip list.
- The snag is that the complexities we obtain will be randomised.
- That is, when we build our data structure, we will do it by tossing coins.
- Then the bounds we will obtain are bounds on the expected number of operations for the worst case input.

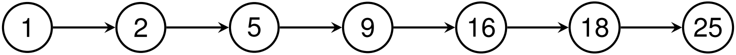
Skip lists

- We will discuss a way in which we can get complexities which match the above AVL tree bounds, with a much simpler data structure, called the skip list.
- The snag is that the complexities we obtain will be randomised.
- That is, when we build our data structure, we will do it by tossing coins.
- Then the bounds we will obtain are bounds on the expected number of operations for the worst case input.
- To be clear: when we perform an Insert, Delete, Find or Successor operation on a skip list, it always succeeds; but sometimes (if we are unlucky with our coin tosses) it might take a long time.

Skip lists

A skip list is a linked list with shortcuts.

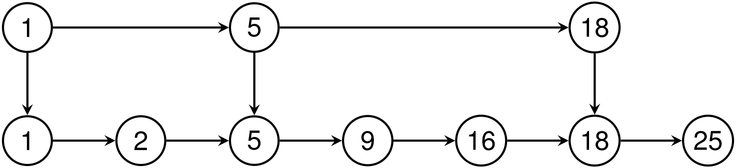
Imagine we have a list containing n keys in sorted order, e.g.:



(data omitted from the diagram for simplicity)

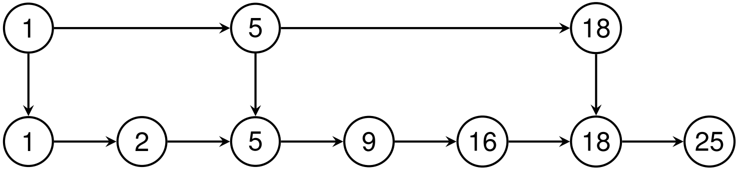
Skip lists

To accelerate search in this list, we attach another linked list which contains duplicates of $m < n$ of the keys in the list, e.g.:



Skip lists

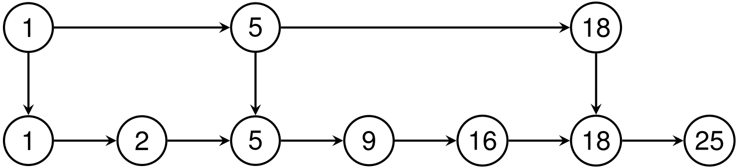
To accelerate search in this list, we attach another linked list which contains duplicates of $m < n$ of the keys in the list, e.g.:



- To find an element, we search in the new "shortcut" list to find the largest key smaller than the key we're looking for.

Skip lists

To accelerate search in this list, we attach another linked list which contains duplicates of $m < n$ of the keys in the list, e.g.:



- To find an element, we search in the new "shortcut" list to find the largest key smaller than the key we're looking for.
- Then we switch to the main list and continue to search for the key as normal.

Optimising this two-level list

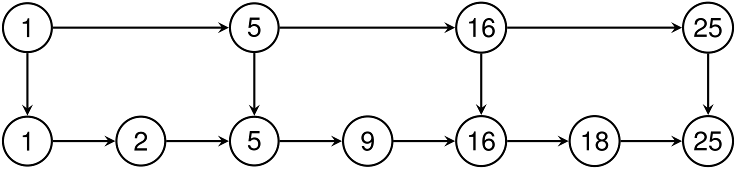
- What is the worst-case time complexity of finding an element in this two-level list?

Optimising this two-level list

- What is the worst-case time complexity of finding an element in this two-level list?
- The number of keys read is the sum of the number read in the main list and the number read in the shortcut list.

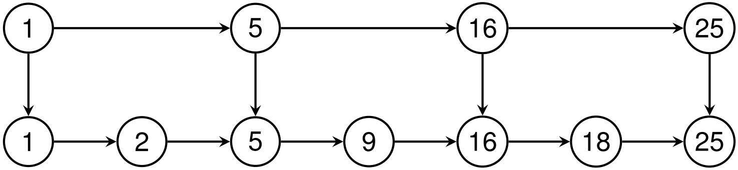
Optimising this two-level list

- What is the worst-case time complexity of finding an element in this two-level list?
- The number of keys read is the sum of the number read in the main list and the number read in the shortcut list.
- A good way to minimise this in the worst case is to make the spacing of the m elements in the shortcut list equal.



Optimising this two-level list

- What is the worst-case time complexity of finding an element in this two-level list?
- The number of keys read is the sum of the number read in the main list and the number read in the shortcut list.
- A good way to minimise this in the worst case is to make the spacing of the m elements in the shortcut list equal.



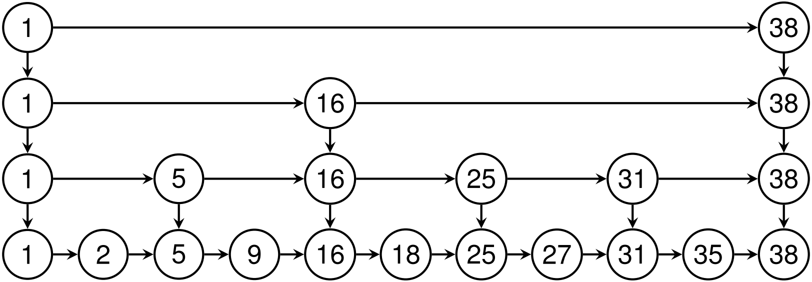
- We then obtain a worst-case complexity of $O(m + n/m)$, which is minimised by taking $m = \sqrt{n}$, giving a complexity of $O(\sqrt{n})$.

Using more levels

- We can improve this complexity by adding more levels to this list, and continuing to keep each level as equally spaced as possible.
- Each element is present in one or more lists, and all elements are present in the bottom list.

Using more levels

- We can improve this complexity by adding more levels to this list, and continuing to keep each level as equally spaced as possible.
- Each element is present in one or more lists, and all elements are present in the bottom list.



To search, we start with the top list and follow the same procedure as with two lists.

Searching in a skip list

Find(k)

1. $i \leftarrow 1$
2. while $i \leq$ number of lists
3. scan along the i'th list until either k is found, or the next element has key greater than k
4. $i \leftarrow i + 1$

Searching in a skip list

Find(k)

1. $i \leftarrow 1$
2. while $i \leq$ number of lists
3. scan along the i'th list until either k is found, or the next element has key greater than k
4. $i \leftarrow i + 1$

- If we have an L-level list, and put $n^{i/L}$ equally spaced elements in each level i between 1 and L, the worst-case number of elements read is

$$\sum_{i=1}^{L} n^{i/L}/n^{(i-1)/L} = \sum_{i=1}^{L} n^{1/L} = Ln^{1/L}.$$

Searching in a skip list

Find(k)

1. $i \leftarrow 1$
2. while $i \leq$ number of lists
3. scan along the i'th list until either k is found, or the next element has key greater than k
4. $i \leftarrow i + 1$

- If we have an L-level list, and put $n^{i/L}$ equally spaced elements in each level i between 1 and L, the worst-case number of elements read is

$$\sum_{i=1}^{L} n^{i/L}/n^{(i-1)/L} = \sum_{i=1}^{L} n^{1/L} = Ln^{1/L}.$$

- If we take $L = \log_2 n$, the total number of elements read is at most $2 \log_2 n$.

Maintaining this data structure

- How should we maintain this structure under insertions and deletions?

Maintaining this data structure

- How should we maintain this structure under insertions and deletions?
- Deletions are easy: when an element is deleted, we remove it from all levels of the list (to make this more efficient, we would use doubly linked lists).

Maintaining this data structure

- How should we maintain this structure under insertions and deletions?
- Deletions are easy: when an element is deleted, we remove it from all levels of the list (to make this more efficient, we would use doubly linked lists).
- Insertion is more tricky. When a new element comes in, we should add it to some number of levels of the list.

Maintaining this data structure

- How should we maintain this structure under insertions and deletions?
- Deletions are easy: when an element is deleted, we remove it from all levels of the list (to make this more efficient, we would use doubly linked lists).
- Insertion is more tricky. When a new element comes in, we should add it to some number of levels of the list.
- The problem is that, as we don't know which elements will arrive in the future, we don't know how many levels of the list to add it to, in order to keep the levels of the list equally spaced.

Maintaining this data structure

- How should we maintain this structure under insertions and deletions?
- Deletions are easy: when an element is deleted, we remove it from all levels of the list (to make this more efficient, we would use doubly linked lists).
- Insertion is more tricky. When a new element comes in, we should add it to some number of levels of the list.
- The problem is that, as we don't know which elements will arrive in the future, we don't know how many levels of the list to add it to, in order to keep the levels of the list equally spaced.
- We will avoid this problem using randomisation.

Maintaining this data structure

Insert(k)

1. search to find where k should be inserted in the bottom level
2. insert k in the bottom level
3. $r \leftarrow$ the result of tossing a fair coin
4. while $r = \textsc{Heads}$
5. insert k in the next level up
6. $r \leftarrow$ the result of tossing a fair coin

Maintaining this data structure

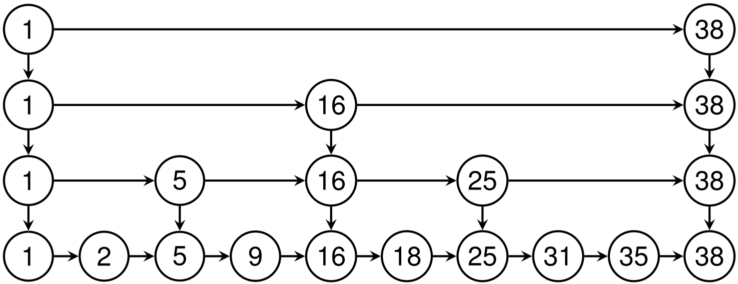
Insert(k)

1. search to find where k should be inserted in the bottom level
2. insert k in the bottom level
3. $r \leftarrow$ the result of tossing a fair coin
4. while $r =$ HEADS
5. insert k in the next level up
6. $r \leftarrow$ the result of tossing a fair coin

So with probability $1/2$, k is only inserted in the main list; with probability $1/4$, it is inserted in the bottom two lists; with probability $1/8$, it is inserted in three lists; ...

Example

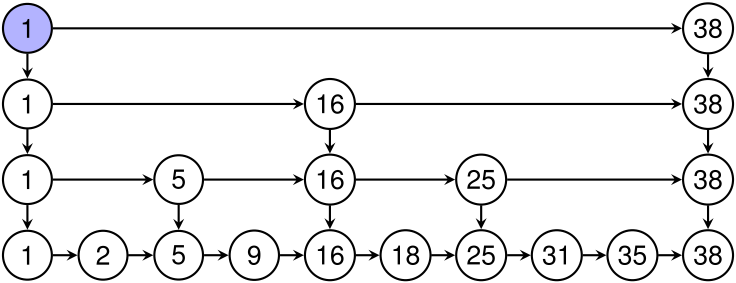
Imagine we start with the following skip list:



Now Insert(27) is called.

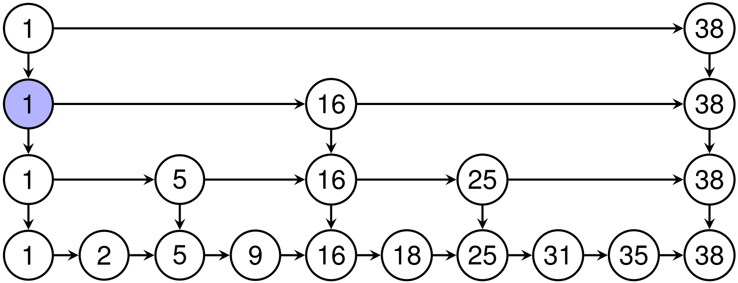
Example

First we search for where 27 should be inserted in the bottom list:



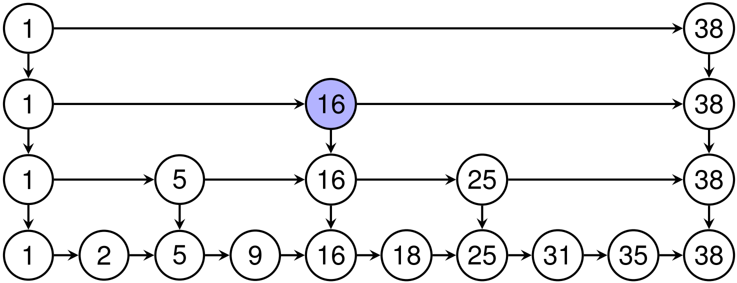
Example

First we search for where 27 should be inserted in the bottom list:



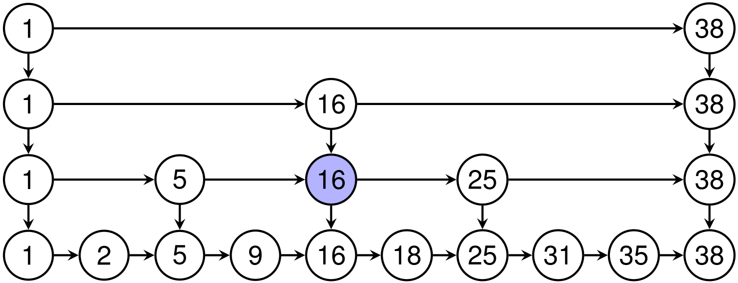
Example

First we search for where 27 should be inserted in the bottom list:



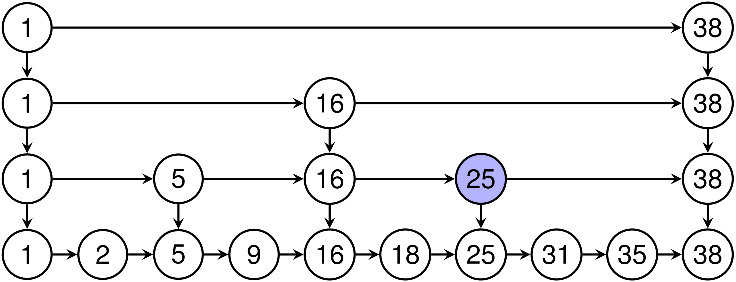
Example

First we search for where 27 should be inserted in the bottom list:



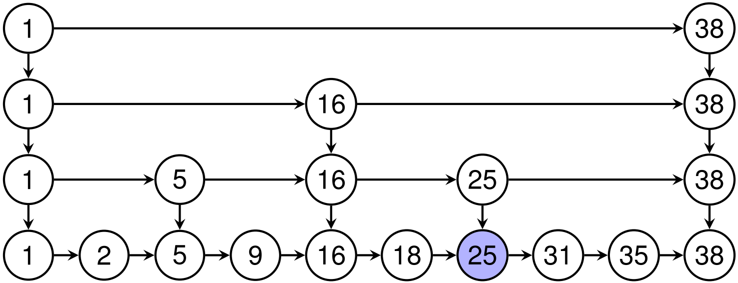
Example

First we search for where 27 should be inserted in the bottom list:



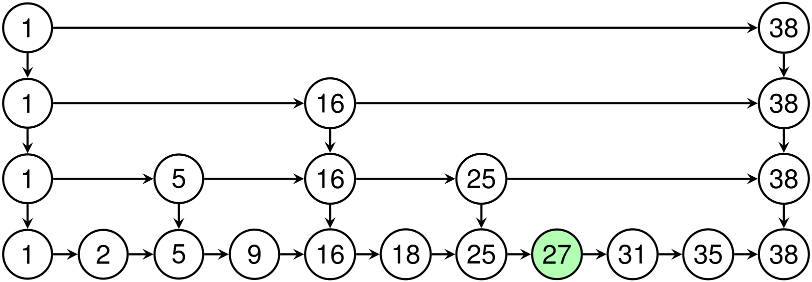
Example

First we search for where 27 should be inserted in the bottom list:



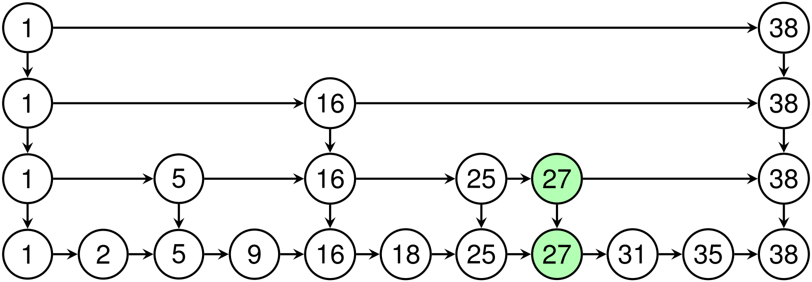
Example

We insert 27 in the bottom list.



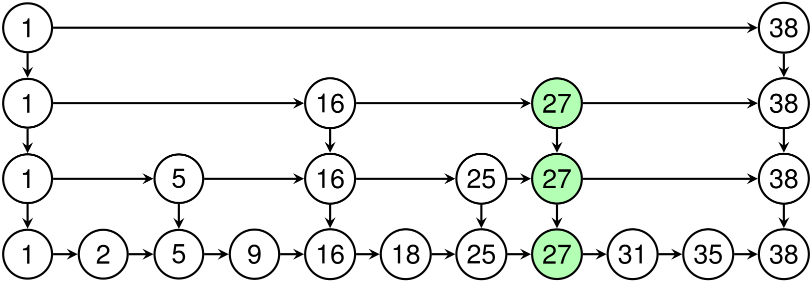
Example

We toss a coin; assume the answer is HEADS. This means we insert 27 in the next level up.



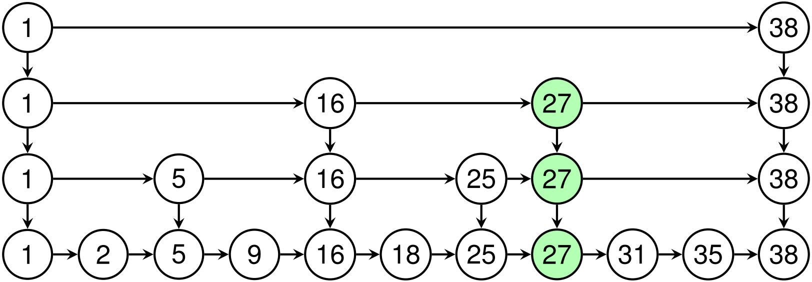
Example

We toss a coin again; assume the answer is HEADS. This means we insert 27 in the next level up too.



Example

We toss a coin again; assume the answer is TAILS. The algorithm terminates.



Probabilistic analysis

- We would like to show that, on average, the skip list has good performance.

Probabilistic analysis

- We would like to show that, on average, the skip list has good performance.
- "On average" should be thought of in terms of the algorithm's internal randomness (coin flips). There is no assumption that the data in the skip list itself is random in any way.

Probabilistic analysis

- We would like to show that, on average, the skip list has good performance.
- "On average" should be thought of in terms of the algorithm's internal randomness (coin flips). There is no assumption that the data in the skip list itself is random in any way.
- Technical note: We assume that the elements to be inserted and deleted are chosen with no reference to the coin flips made by the algorithm.

Probabilistic analysis

- We would like to show that, on average, the skip list has good performance.
- "On average" should be thought of in terms of the algorithm's internal randomness (coin flips). There is no assumption that the data in the skip list itself is random in any way.
- Technical note: We assume that the elements to be inserted and deleted are chosen with no reference to the coin flips made by the algorithm.

The main tool from probability theory we will need:

Union bound

Let $E_1, \dots, E_m$ be events, and let the probability of event E_i occurring be p_i. Then the probability that one or more of the events E_1 through E_m occur is at most $\sum_i p_i$.

The number of levels

We first show that a skip list does not have many levels, with high probability.

Claim

The probability that a skip list containing n elements has $2\log_2 n$ levels or more is at most $1/n$.

The number of levels

We first show that a skip list does not have many levels, with high probability.

Claim

The probability that a skip list containing n elements has $2 \log_2 n$ levels or more is at most $1/n$.

Proof

- By the union bound, the probability of having at least $2 \log_2 n$ levels is at most n times the probability that an individual element is inserted in at least $2 \log_2 n$ levels.

The number of levels

We first show that a skip list does not have many levels, with high probability.

Claim

The probability that a skip list containing n elements has $2\log_2 n$ levels or more is at most $1/n$.

Proof

- By the union bound, the probability of having at least $2\log_2 n$ levels is at most n times the probability that an individual element is inserted in at least $2\log_2 n$ levels.
- This probability is precisely $(1/2)^{2\log_2 n} = 1/n^2$.

The number of levels

We first show that a skip list does not have many levels, with high probability.

Claim

The probability that a skip list containing n elements has $2\log_2 n$ levels or more is at most $1/n$.

Proof

- By the union bound, the probability of having at least $2\log_2 n$ levels is at most n times the probability that an individual element is inserted in at least $2\log_2 n$ levels.
- This probability is precisely $(1/2)^{2\log_2 n} = 1/n^2$.
- So the probability that the list has at least $2\log_2 n$ levels is at most $n \times 1/n^2 = 1/n$.

$\square$

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- We analyse the behaviour of the algorithm when searching for an item.

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- We analyse the behaviour of the algorithm when searching for an item.
- Key observation: apart from the first item, we only examine an element at a given level if it was not present on the level above.

Ashley Montanaro
ashley@cs.bris.ac.uk

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- We analyse the behaviour of the algorithm when searching for an item.
- Key observation: apart from the first item, we only examine an element at a given level if it was not present on the level above.
- Therefore, the expected number of elements examined on a given level is the same as the expected number of flips of a coin required until we get heads.

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- We analyse the behaviour of the algorithm when searching for an item.
- Key observation: apart from the first item, we only examine an element at a given level if it was not present on the level above.
- Therefore, the expected number of elements examined on a given level is the same as the expected number of flips of a coin required until we get heads.
- This is $1/2 \times 1 + 1/4 \times 2 + 1/8 \times 3 \cdots = \sum_{i \geq 1} i2^{-i} = 2$.

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- We analyse the behaviour of the algorithm when searching for an item.
- Key observation: apart from the first item, we only examine an element at a given level if it was not present on the level above.
- Therefore, the expected number of elements examined on a given level is the same as the expected number of flips of a coin required until we get heads.
- This is $1/2 \times 1 + 1/4 \times 2 + 1/8 \times 3 \cdots = \sum_{i \geq 1} i2^{-i} = 2$.
- So if there are at most $2 \log_2 n$ levels in the list, we examine an expected number of at most $4 \log_2 n$ elements.

...

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- On the other hand, we always examine at most n elements.

The search time

Claim

Let L be a skip list containing n elements. Then the expected time to find an element in L is $O(\log n)$.

Proof (sketch)

- On the other hand, we always examine at most n elements.
- So the expected number of elements examined is at most

$$\begin{aligned} & \Pr[\geq 2\log_2 n \text{ levels}] \times n + \Pr[\leq 2\log_2 n \text{ levels}] \times 2 \times (2\log_2 n) \\ \leq\ & 1/n \times n + 1 \times 2 \times (2\log_2 n) \\ =\ & O(\log n). \end{aligned}$$

□

Summary of skip lists

- A skip list is a linked list with shortcuts. The skip list shows how randomness can be useful in the design of data structures.

Summary of skip lists

- A skip list is a linked list with shortcuts. The skip list shows how randomness can be useful in the design of data structures.

- This is possibly surprising as we might imagine that randomness is the last thing we would want to build into data storage.

Summary of skip lists

- A skip list is a linked list with shortcuts. The skip list shows how randomness can be useful in the design of data structures.
- This is possibly surprising as we might imagine that randomness is the last thing we would want to build into data storage.
- Skip lists achieve similar performance to AVL trees and other balanced binary trees, but are simpler and more intuitive to implement.

Summary of skip lists

- A skip list is a linked list with shortcuts. The skip list shows how randomness can be useful in the design of data structures.
- This is possibly surprising as we might imagine that randomness is the last thing we would want to build into data storage.
- Skip lists achieve similar performance to AVL trees and other balanced binary trees, but are simpler and more intuitive to implement.
- They are randomised, so in theory could have bad worst-case performance, but in practice have excellent performance.

Summary of skip lists

- A skip list is a linked list with shortcuts. The skip list shows how randomness can be useful in the design of data structures.
- This is possibly surprising as we might imagine that randomness is the last thing we would want to build into data storage.
- Skip lists achieve similar performance to AVL trees and other balanced binary trees, but are simpler and more intuitive to implement.
- They are randomised, so in theory could have bad worst-case performance, but in practice have excellent performance.
- A number of advanced database applications are built around skip lists, e.g. levelDB (Google).

Other search structures?

- Another interesting data structure, which achieves similar performance to skip lists, is the treap.

Other search structures?

- Another interesting data structure, which achieves similar performance to skip lists, is the treap.

- Just as skip lists can be seen as randomised linked lists, treaps can be seen as randomised binary trees.

Other search structures?

- Another interesting data structure, which achieves similar performance to skip lists, is the treap.

- Just as skip lists can be seen as randomised linked lists, treaps can be seen as randomised binary trees.

- On modern computer systems, there are many other factors to be considered when choosing a search structure (e.g. performance with respect to caching and external memory, concurrent access, . . .).

Summary of complexities

Structure	Space	Insert	Delete	Find	Successor
Array	$\Theta(U)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(U)$
Unsorted linked list	$O(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(n)$
Hash table	$O(n)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(n)$
Binary tree	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Skip list	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Holy grail	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

- All complexities listed are worst-case.
- The skip list is randomised; all others are deterministic.
- How close can we get to the holy grail? An object of current research! See COMS31900: Advanced Algorithms for more...

Further Reading

- Introduction to Algorithms
  T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein.
  MIT Press/McGraw-Hill, ISBN: 0-262-03293-7.
  - Section 10.2 – Linked lists
  - Section 11.1 – Directly addressed arrays
  - Section 11.2 – Hash tables
  - Exercise 13-3 – AVL trees
  - Exercise 13-4 – Treaps

- Algorithms lecture notes, University of Illinois
  Jeff Erickson
  `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/`
  - Lecture 10 – Treaps and skip lists

Biographical notes

William J. Pugh

- Bill Pugh developed skip lists in the 1980s.
- Also contributions to programming language design and implementation, including the Java memory model.
- Currently a professor emeritus at the University of Maryland.

Pic: umd.edu

Ashley Montanaro
ashley@cs.bris.ac.uk