Dynamic Search Structures

Self-balancing Trees and Skip Lists

Benjamin Sach
Dynamic Search Structures

A **dynamic search structure**, stores a set of elements

*Each element* \( x \) *must have a unique key* - \( x.key \)

The following operations are supported:

- \( \text{INSERT}(x, k) \) - inserts \( x \) with key \( k = x.key \)
- \( \text{FIND}(k) \) - returns the (unique) element \( x \) with \( x.key = k \) \( \text{(or reports that it doesn't exist)} \)
- \( \text{DELETE}(k) \) - deletes the (unique) element \( x \) with \( x.key = k \) \( \text{(or reports that it doesn't exist)} \)
Dynamic Search Structures

A dynamic search structure, stores a set of elements

*Each element* $x$ *must have a unique key* - $x.key$

The following operations are supported:

$$\text{INSERT}(x, k)$$ - inserts $x$ with key $k = x.key$

$$\text{FIND}(k)$$ - returns the (unique) element $x$ with $x.key = k$

(\text{or reports that it doesn't exist})

$$\text{DELETE}(k)$$ - deletes the (unique) element $x$ with $x.key = k$

(\text{or reports that it doesn't exist})

We would also like it to support (among others):

$$\text{PREDECESSOR}(k)$$ - returns the (unique) element $x$

with the largest key such that $x.key < k$

$$\text{RANGEFIND}(k_1, k_2)$$ - returns every element $x$ with $k_1 \leq x.key \leq k_2$$
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren't all efficient

Let \( n \) denote the number of elements stored in the structure

- our goal is to implement a structure with operations which scale well as \( n \) grows
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let $n$ denote the number of elements stored in the structure
- our goal is to implement a structure with operations which scale well as $n$ grows

We could implement a Dynamic Search Structure using an unsorted linked list:
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let \( n \) denote the number of elements stored in the structure

- our goal is to implement a structure with operations which scale well as \( n \) grows

We could implement a Dynamic Search Structure using an unsorted linked list:
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let $n$ denote the number of elements stored in the structure
- our goal is to implement a structure with operations which scale well as $n$ grows

We could implement a Dynamic Search Structure using an unsorted linked list:

INSERT is very efficient,
- add the new item to the head of the list in $O(1)$ time
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let $n$ denote the number of elements stored in the structure

- our goal is to implement a structure with operations which scale well as $n$ grows

We could implement a Dynamic Search Structure using an unsorted linked list:

- INSERT is very efficient,
  - add the new item to the head of the list in $O(1)$ time
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let \( n \) denote the number of elements stored in the structure

- our goal is to implement a structure with operations which scale well as \( n \) grows

We could implement a Dynamic Search Structure using an unsorted linked list:

\[
\begin{array}{c}
\text{Chris} & \xrightarrow{7} & \text{Emma} & \xrightarrow{6} & \text{Bob} & \xrightarrow{5} & \text{Dawn} & \xrightarrow{4} & \text{Alice} & \xrightarrow{3}
\end{array}
\]

**INSERT** is very efficient,

- add the new item to the head of the list in \( O(1) \) time

**FIND** and **DELETE** are very inefficient, they take \( O(n) \) time

- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren't all efficient

Let \( n \) denote the number of elements stored in the structure
- our goal is to implement a structure with operations which scale well as \( n \) grows

We could implement a Dynamic Search Structure using an unsorted linked list:

\[
\begin{array}{c}
\text{Chris} \\
7
\end{array} \quad \begin{array}{c}
\text{Emma} \\
6
\end{array} \quad \begin{array}{c}
\text{Bob} \\
5
\end{array} \quad \begin{array}{c}
\text{Dawn} \\
4
\end{array} \quad \begin{array}{c}
\text{Alice} \\
3
\end{array}
\]

\( \text{INSERT} \) is very efficient,
- add the new item to the head of the list in \( O(1) \) time

\( \text{FIND} \) and \( \text{DELETE} \) are very inefficient, they take \( O(n) \) time
- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let \( n \) denote the number of elements stored in the structure
- our goal is to implement a structure with operations which scale well as \( n \) grows

We could implement a Dynamic Search Structure using an unsorted linked list:

\[
\begin{array}{c}
\text{Chris} & \text{Emma} & \text{Bob} & \text{Dawn} & \text{Alice} \\
7 & 6 & 5 & 4 & 3
\end{array}
\]

**INSERT** is very efficient,
- add the new item to the head of the list in \( O(1) \) time

**FIND** and **DELETE** are very *inefficient*, they take \( O(n) \) time
- we have to look through the entire linked list to find an item
  \((\text{in the worst case})\)
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren't all efficient

Let $n$ denote the number of elements stored in the structure
- our goal is to implement a structure with operations which scale well as $n$ grows

We could implement a Dynamic Search Structure using an unsorted linked list:

**INSERT** is very efficient,
- add the new item to the head of the list in $O(1)$ time

**FIND** and **DELETE** are very inefficient, they take $O(n)$ time
- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... *but they aren’t all efficient*

Let $n$ denote the number of elements stored in the structure

- *our goal is to implement a structure with operations which scale well as $n$ grows*

We could implement a Dynamic Search Structure using an unsorted linked list:

```
Chris  7  →  Emma  6  →  Bob  5  →  Dawn  4  →  Alice  3
```

**INSERT** is very efficient,
- add the new item to the head of the list in $O(1)$ time

**FIND** and **DELETE** are very *inefficient*, they take $O(n)$ time
- we have to look through the entire linked list to find an item *(in the worst case)*
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let \( n \) denote the number of elements stored in the structure

- our goal is to implement a structure with operations which scale well as \( n \) grows

We could implement a Dynamic Search Structure using an unsorted linked list:

![Linked List Diagram]

**INSERT** is very efficient,

- add the new item to the head of the list in \( O(1) \) time

**FIND** and **DELETE** are very inefficient, they take \( O(n) \) time

- we have to look through the entire linked list to find an item (in the worst case)
Using a Linked List as a Dynamic Search Structure

There are many ways in which we could implement a search structure... but they aren’t all efficient

Let $n$ denote the number of elements stored in the structure
- *our goal is to implement a structure with operations which scale well as $n$ grows*

We could implement a Dynamic Search Structure using an unsorted linked list:

**INSERT** is very efficient,
- add the new item to the head of the list in $O(1)$ time

**FIND** and **DELETE** are very *inefficient*, they take $O(n)$ time
- we have to look through the entire linked list to find an item *(in the worst case)*
A classic choice for a dynamic search structure is a binary search tree…
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree…

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
A classic choice for a dynamic search structure is a binary search tree…

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys.
- all the nodes in the right subtree have larger keys.

We perform a `FIND` operation by following a path from the root.
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

21 < 27 so go left

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree... 

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a `FIND` operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node:

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... 

this takes $O(h)$ time - where $h$ is the height
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a \textbf{FIND} operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The \textbf{INSERT} and \textbf{DELETE} operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **Find** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **Insert** and **Delete** operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

21 < 27 so go left

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The INSERT and DELETE operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root...
this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

21 < 27 so go left
21 > 20 so go right
21 < 23 so go left

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes \( O(h) \) time - where \( h \) is the height

The **INSERT** and **DELETE** operations are similar and also take \( O(h) \) time
A classic choice for a dynamic search structure is a binary search tree…

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

21 < 27 so go left
21 > 20 so go right
21 < 23 so go left
21 = 21 found it!

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...

this takes $O(h)$ time - where $h$ is the height

The INSERT and DELETE operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **Find** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **Insert** and **Delete** operations are similar and also take $O(h)$ time

21 < 27 so go left
21 > 20 so go right
21 < 23 so go left
21 = 21 found it!
now delete it!

```
DELETE(21)
```
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
this takes $O(h)$ time - where $h$ is the height

The INSERT and DELETE operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a \texttt{FIND} operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The \texttt{INSERT} and \texttt{DELETE} operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node

- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **Find** operation by following a path from the root...

this takes $O(h)$ time - where $h$ is the height

The **Insert** and **Delete** operations are similar and also take $O(h)$ time
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
this takes $O(h)$ time - where $h$ is the height

The INSERT and DELETE operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a **FIND** operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The **INSERT** and **DELETE** operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree...

Recall that in a binary search tree, for any node:
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root... this takes $O(h)$ time - where $h$ is the height

The INSERT and DELETE operations are similar and also take $O(h)$ time
A classic choice for a dynamic search structure is a binary search tree.

Recall that in a binary search tree, for any node
- all the nodes in the left subtree have smaller keys
- all the nodes in the right subtree have larger keys

We perform a FIND operation by following a path from the root...
this takes $O(h)$ time - where $h$ is the height

The INSERT and DELETE operations are similar and also take $O(h)$ time
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree…
A classic choice for a dynamic search structure is a binary search tree...

How big is $h$?
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree…

\( n \) is the number of elements

How big is \( h \)?
A classic choice for a dynamic search structure is a binary search tree…

\( n \) is the number of elements

How big is \( h \)?

It might be as small as \( \log_2 n \) (if the tree is perfectly balanced)
A classic choice for a dynamic search structure is a binary search tree...

$n$ is the number of elements

How big is $h$?

It might be as small as $\log_2 n$ (if the tree is perfectly balanced)

but it might be as big as $n$ (if the tree is completely unbalanced)
Binary Search Trees

A classic choice for a dynamic search structure is a binary search tree...

$n$ is the number of elements

How big is $h$?

It might be as small as $\log_2 n$ (if the tree is perfectly balanced)
but it might be as big as $n$ (if the tree is completely unbalanced)

In particular, each \text{INSERT} could increase $h$ by one
A classic choice for a dynamic search structure is a binary search tree...

\( n \) is the number of elements

How big is \( h \)?

It might be as small as \( \log_2 n \) (if the tree is perfectly balanced)

but it might be as big as \( n \) (if the tree is completely unbalanced)

In particular, each \texttt{INSERT} could increase \( h \) by one
A classic choice for a dynamic search structure is a binary search tree...

\( n \) is the number of elements

How big is \( h \)?

It might be as small as \( \log_2 n \) (if the tree is perfectly balanced)

but it might be as big as \( n \) (if the tree is completely unbalanced)

In particular, each \texttt{INSERT} could increase \( h \) by one
A classic choice for a dynamic search structure is a binary search tree...

\[ n \text{ is the number of elements} \]

How big is \( h \)?

It might be as small as \( \log_2 n \) (if the tree is perfectly balanced)

but it might be as big as \( n \) (if the tree is completely unbalanced)

In particular, each \texttt{INSERT} could increase \( h \) by one
A classic choice for a dynamic search structure is a binary search tree…

\[ n \text{ is the number of elements} \]

How big is \( h \)?

It might be as small as \( \log_2 n \) (if the tree is perfectly balanced)

but it might be as big as \( n \) (if the tree is completely unbalanced)

In particular, each \textbf{INSERT} could increase \( h \) by one

\textit{how can we overcome this?}
Part one
Self-balancing trees

*inspired by slides by Inge Li Gørtz
in turn inspired by slides by Kevin Wayne*
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

2-node: 2 children and 1 key

3-node: 3 children and 2 keys

4-node: 4 children and 3 keys
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

| 2-node: | 2 children and 1 key |
| 3-node: | 3 children and 2 keys |
| 4-node: | 4 children and 3 keys |
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

<table>
<thead>
<tr>
<th>2-node: 2 children and 1 key</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-node: 3 children and 2 keys</td>
</tr>
<tr>
<td>4-node: 4 children and 3 keys</td>
</tr>
</tbody>
</table>
2-3-4 Trees

Key idea: Nodes can have between 2 and 4 children (hence the name)

Perfect balance - every path from the root to a leaf has the same length (always, all the time)

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

- **2-node:** 2 children and 1 key
- **3-node:** 3 children and 2 keys
- **4-node:** 4 children and 3 keys
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

---

**Nodes:**
- **2-node:** 2 children and 1 key
- **3-node:** 3 children and 2 keys
- **4-node:** 4 children and 3 keys

*The ● are “dummy leaves” (they don’t do or contain anything)*
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys

*Like in a binary search tree, the keys held at a node determine the contents of its subtrees*

*The ● are “dummy leaves” (they don’t do or contain anything)*
2-3-4 Trees

Key idea: Nodes can have between 2 and 4 children (hence the name)

Perfect balance - every path from the root to a leaf has the same length (always, all the time)

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys

The • are “dummy leaves” (they don’t do or contain anything)
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

Like in a binary search tree, *the keys held at a node determine the contents of its subtrees*

*The ● are “dummy leaves” (they don’t do or contain anything)*
2-3-4 Trees

Key idea: Nodes can have between 2 and 4 children (hence the name)

Perfect balance - every path from the root to a leaf has the same length (always, all the time)

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The ● are “dummy leaves” (they don’t do or contain anything)
**2-3-4 Trees**

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The black circles are “dummy leaves” (they don’t do or contain anything)
**2-3-4 Trees**

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

- **2-node:** 2 children and 1 key
- **3-node:** 3 children and 2 keys
- **4-node:** 4 children and 3 keys

*Like in a binary search tree, the keys held at a node determine the contents of its subtrees*

*The ● are “dummy leaves” (they don’t do or contain anything)*
2-3-4 Trees

Key idea: Nodes can have between 2 and 4 children (hence the name)

Perfect balance - every path from the root to a leaf has the same length (always, all the time)

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The ⦿ are “dummy leaves” (they don’t do or contain anything)

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The ● are “dummy leaves” (they don’t do or contain anything)
2-3-4 Trees

Key idea: Nodes can have between 2 and 4 children (hence the name)

Perfect balance - every path from the root to a leaf has the same length (always, all the time)

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The ● are “dummy leaves” (they don’t do or contain anything)
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

2-node: 2 children and 1 key
3-node: 3 children and 2 keys
4-node: 4 children and 3 keys

Like in a binary search tree, *the keys held at a node determine the contents of its subtrees*

The ● are “dummy leaves” (they don’t do or contain anything)
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

- **2-node:** 2 children and 1 key
- **3-node:** 3 children and 2 keys
- **4-node:** 4 children and 3 keys

*Like in a binary search tree,* the keys held at a node determine the contents of its subtrees

*The ● are “dummy leaves” (they don’t do or contain anything)*
**2-3-4 Trees**

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

---

**2-node:** 2 children and 1 key  
**3-node:** 3 children and 2 keys  
**4-node:** 4 children and 3 keys

Like in a binary search tree,  
*the keys held at a node determine the contents of its subtrees*

*The ○ are “dummy leaves” (they don’t do or contain anything)*
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children *(hence the name)*

**Perfect balance** - every path from the root to a leaf has the same length *(always, all the time)*

*Like in a binary search tree, the keys held at a node determine the contents of its subtrees*

**2-node:** 2 children and 1 key
**3-node:** 3 children and 2 keys
**4-node:** 4 children and 3 keys

*The ○ are “dummy leaves” (they don’t do or contain anything)*
2-3-4 Trees

Key idea: Nodes can have between 2 and 4 children (hence the name)

Perfect balance - every path from the root to a leaf has the same length (always, all the time)

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The ● are “dummy leaves” (they don’t do or contain anything)
2-3-4 Trees

**Key idea:** Nodes can have between 2 and 4 children (hence the name)

**Perfect balance** - every path from the root to a leaf has the same length (always, all the time)

Like in a binary search tree, the keys held at a node determine the contents of its subtrees

The ● are “dummy leaves” (they don’t do or contain anything)

- **2-node:** 2 children and 1 key
- **3-node:** 3 children and 2 keys
- **4-node:** 4 children and 3 keys
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root.

*Decisions are made by inspecting the key(s) at the current node and following the appropriate edge.*
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...
The **FIND** operation

Just like in a binary search tree,
we perform a **FIND** operation by following a path from the root...

decisions are made by inspecting the key(s) at the current node
and following the appropriate edge
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18

\[ \text{FIND}(12) \]

**descisions are made by inspecting the key(s) at the current node and following the appropriate edge**
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18

**FIND**(12)

decisions are made by inspecting the key(s) at the current node and following the appropriate edge
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18

```
1  3  5  7  10  12  14  17  19  22  24  25  26
  2  4  6  
    11  18
```

**descisions are made by inspecting the key(s) at the current node and following the appropriate edge**
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13

*decisions are made by inspecting the key(s) at the current node and following the appropriate edge*
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18  
12 is smaller than 13

**FIND**(12)

descisions are made by inspecting the key(s) at the current node and following the appropriate edge
The **FIND** operation

Just like in a binary search tree,
we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13

**descisions are made by inspecting the key(s) at the current node**
**and following the appropriate edge**
The **FIND** operation

Just like in a binary search tree,
we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13
found it!

`FIND(12)`

decisions are made by inspecting the key(s) at the current node
and following the appropriate edge
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13
found it!

decisions are made by inspecting the key(s) at the current node and following the appropriate edge

What is the time complexity of the **FIND** operation?
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13
found it!

**FIND(12)**

Decisions are made by inspecting the key(s) at the current node and following the appropriate edge

What is the time complexity of the **FIND** operation?

It’s $O(h)$ again
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13
found it!

**FIND**(12)

\[
\begin{array}{c}
2 & 4 & 6 \\
| & & |
\end{array}
\quad
\begin{array}{c}
11 & 18 \\
| & |
\end{array}
\quad
\begin{array}{c}
13 & 15 \\
| & |
\end{array}
\quad
\begin{array}{c}
24 \\
|
\end{array}
\]

\[
\begin{array}{c}
1 & 3 & 5 \\
| & & |
\end{array}
\quad
\begin{array}{c}
7 & 10 \\
| & |
\end{array}
\quad
\begin{array}{c}
12 \\
|
\end{array}
\quad
\begin{array}{c}
14 & 17 \\
| & |
\end{array}
\quad
\begin{array}{c}
19 & 22 \\
| & |
\end{array}
\quad
\begin{array}{c}
25 & 26 \\
|
\end{array}
\]

**descisions are made by inspecting the key(s) at the current node**
and following the appropriate edge

What is the time complexity of the **FIND** operation?

It’s \( O(h) \) again

(each step down the path takes \( O(1) \) time)
The **FIND** operation

Just like in a binary search tree, we perform a **FIND** operation by following a path from the root...

12 is between 11 and 18
12 is smaller than 13
found it!

descisions are made by inspecting the key(s) at the current node and following the appropriate edge

What is the time complexity of the **FIND** operation?

It’s $O(h)$ again

(each step down the path takes $O(1)$ time)

What is the height, $h$ of a 2-3-4 tree?
The height of a 2-3-4 tree

**Perfect balance** - every path from the root to a leaf has the same length

(we’ll justify this as we go along)

This implies that the height, $h$ of a 2-3-4 tree with $n$ nodes is

Best case: $\log_4 n = \frac{\log_2 n}{2}$ (all 4-nodes)

Worst case: $\log_2 n$ (all 2-nodes)

$h$ is between 10 and 20 for a million nodes
The height of a 2-3-4 tree

Perfect balance - every path from the root to a leaf has the same length (we’ll justify this as we go along)

This implies that the height, \( h \) of a 2-3-4 tree with \( n \) nodes is

Best case: \( \log_4 n = \frac{\log_2 n}{2} \) (all 4-nodes)

Worst case: \( \log_2 n \) (all 2-nodes)

\( h \) is between 10 and 20 for a million nodes

The time complexity of the FIND operation is \( O(h) \)
The height of a 2-3-4 tree

**Perfect balance** - every path from the root to a leaf has the same length

(we’ll justify this as we go along)

This implies that the height, \( h \) of a 2-3-4 tree with \( n \) nodes is

Best case: \( \log_4 n = \frac{\log_2 n}{2} \) (all 4-nodes)

Worst case: \( \log_2 n \) (all 2-nodes)

\( h \) is between 10 and 20 for a million nodes

The time complexity of the **FIND** operation is \( O(h) = O(\log n) \)
The **INSERT** operation

To perform $\text{INSERT}(x, k)$,
The **INSERT** operation

To perform $\text{INSERT}(x, k)$,

**Step 1:** Search for the key $k$ as if performing $\text{FIND}(k)$. 
To perform \( \text{INSERT}(x, k) \),

**Step 1**: Search for the key \( k \) as if performing \( \text{FIND}(k) \).
To perform \textsc{Insert}(x, k),

**Step 1:** Search for the key \( k \) as if performing \textsc{Find}(k).
To perform $\text{INSERT}(x, k)$,

**Step 1:** Search for the key $k$ as if performing $\text{FIND}(k)$. 
The **INSERT** operation

To perform INSERT\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing FIND\((k)\).
The **INSERT** operation

To perform **INSERT**(x, k),

**Step 1:** Search for the key k as if performing **FIND**(k).

**Step 2:** If the leaf is a 2-node, insert (x, k), converting it into a 3-node.
To perform $\text{INSERT}(x, k)$,

**Step 1:** Search for the key $k$ as if performing $\text{FIND}(k)$.

**Step 2:** If the leaf is a 2-node, insert $(x, k)$, converting it into a 3-node.
The **INSERT** operation

To perform \texttt{INSERT}(x, k),

**Step 1:** Search for the key \(k\) as if performing \texttt{FIND}(k).

**Step 2:** If the leaf is a 2-node, 
insert \((x, k)\), converting it into a 3-node.
To perform \textsc{insert}(x, k),

\textbf{Step 1:} Search for the key \( k \) as if performing \textsc{find}(k).

\textbf{Step 2:} If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node
To perform $\text{INSERT}(x, k)$,

**Step 1:** Search for the key $k$ as if performing $\text{FIND}(k)$.

**Step 2:** If the leaf is a 2-node, insert $(x, k)$, converting it into a 3-node.
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node
The \textbf{INSERT} operation

To perform $\text{INSERT}(x, k)$,

\textbf{Step 1:} Search for the key $k$ as if performing $\text{FIND}(k)$.

\textbf{Step 2:} If the leaf is a 2-node,
insert $(x, k)$, converting it into a 3-node.
The **INSERT** operation

To perform **INSERT**(\(x, k\)),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node
The \textbf{INSERT} operation

To perform \texttt{INSERT}(x, k),

\textbf{Step 1:} Search for the key \(k\) as if performing \texttt{FIND}(k).

\textbf{Step 2:} If the leaf is a 2-node,
\begin{itemize}
  \item insert \((x, k)\), converting it into a 3-node
\end{itemize}

\textbf{Step 3:} If the leaf is a 3-node,
\begin{itemize}
  \item insert \((x, k)\), converting it into a 4-node
\end{itemize}
The **INSERT** operation

To perform \( \text{INSERT}(x, k) \),

**Step 1:** Search for the key \( k \) as if performing **FIND**(\( k \)).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**(*x*, *k*),

**Step 1:** Search for the key *k* as if performing **FIND**(*k*).

**Step 2:** If the leaf is a 2-node,

insert (*x*, *k*), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert (*x*, *k*), converting it into a 4-node
The **INSERT** operation

$$\text{INSERT}(x, 8)$$

To perform \text{INSERT}(x, k),

**Step 1:** Search for the key $k$ as if performing \text{FIND}(k).

**Step 2:** If the leaf is a 2-node,

insert $(x, k)$, converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert $(x, k)$, converting it into a 4-node
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node, insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node, insert \((x, k)\), converting it into a 4-node
To perform $\text{INSERT}(x, k)$,

**Step 1:** Search for the key $k$ as if performing $\text{FIND}(k)$.

**Step 2:** If the leaf is a 2-node,
insert $(x, k)$, converting it into a 3-node

**Step 3:** If the leaf is a 3-node,
insert $(x, k)$, converting it into a 4-node
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node,
insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,
insert \((x, k)\), converting it into a 4-node

**Step 4:** If the leaf is a 4-node,
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node

**Step 4:** If the leaf is a 4-node,  ???
The **INSERT** operation

To perform **INSERT**(*x*, *k*),

**Step 1:** Search for the key *k* as if performing **FIND**(*k*).

**Step 2:** If the leaf is a 2-node,

insert (*x*, *k*), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert (*x*, *k*), converting it into a 4-node

**Step 4:** If the leaf is a 4-node,  

We will make sure this *never* happens
**Splitting 4-nodes**

We can split any 4-node into two 2-nodes if it's parent isn't a 4-node.
We can split any 4-node into two 2-nodes if it's parent isn't a 4-node.
**Splitting 4-nodes**

We can split any 4-node into two 2-nodes if its parent isn’t a 4-node.

**BEFORE**

```
  41 63
  /   \
32   51
```

**AFTER**

```
  41 63 86
  /   /   \
32  51  71 92
```

The extra key is pushed up to the parent (so it won’t work if the parent is a 4-node).
**SPLITTING 4-nodes**

We can **SPLIT** any 4-node into two 2-nodes if its parent isn’t a 4-node.

**BEFORE**

these subtrees could have any size

**AFTER**

The extra key is pushed up to the parent (so it won’t work if the parent is a 4-node)
**SPLITTING 4-nodes**

We can split any 4-node into two 2-nodes if it’s parent isn’t a 4-node.

**BEFORE**

```
32  51  71  86  92
```

*these subtrees could have any size*

**AFTER**

```
32  51  71  92
```

*The extra key is pushed up to the parent (so it won’t work if the parent is a 4-node)*

```
41  63  86
```

*these subtrees haven’t changed*
**SPLITTING 4-nodes**

We can **SPLIT** any 4-node into two 2-nodes if its parent isn’t a 4-node.

**BEFORE**

- These subtrees could have any size.

**AFTER**

- The extra key is pushed up to the parent (so it won’t work if the parent is a 4-node).
- These subtrees haven’t changed.
- No path lengths have changed.
We can **split** any 4-node into two 2-nodes if its parent isn’t a 4-node.

These subtrees could have any size.

The extra key is pushed up to the parent (so it won’t work if the parent is a 4-node).

No path lengths have changed.

(If it was **perfectly balanced**, it still is.)
**Splitting 4-nodes**

We can **split** any 4-node into two 2-nodes if it’s parent isn’t a 4-node.

**Before**

![Before diagram]

- These subtrees could have any size.
- The extra key is pushed up to the parent (so it won’t work if the parent is a 4-node).
- No path lengths have changed (if it was perfectly balanced, it still is).

**After**

![After diagram]

- These subtrees haven’t changed.

**Split** takes $O(1)$ time.
To perform $\text{INSERT}(x, k)$,

**Step 1:** Search for the key $k$ as if performing $\text{FIND}(k)$.

**Step 2:** If the leaf is a 2-node,

insert $(x, k)$, converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert $(x, k)$, converting it into a 4-node
The **INSERT** operation

To perform **INSERT**($x, k$),

**Step 1**: Search for the key $k$ as if performing **FIND**($k$).

**Step 2**: If the leaf is a 2-node,

- insert ($x, k$), converting it into a 3-node

**Step 3**: If the leaf is a 3-node,

- insert ($x, k$), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**(*x*, *k*),

**Step 1:** Search for the key *k* as if performing **FIND**(*k*).

**Step 2:** If the leaf is a 2-node,

insert (*x*, *k*), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert (*x*, *k*), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node, insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node, insert \((x, k)\), converting it into a 4-node
To perform $\text{INSERT}(x, k)$,

**Step 1**: Search for the key $k$ as if performing $\text{FIND}(k)$.

**Step 2**: If the leaf is a 2-node,

- insert $(x, k)$, converting it into a 3-node

**Step 3**: If the leaf is a 3-node,

- insert $(x, k)$, converting it into a 4-node

*Split 4-nodes as we go down*
The **INSERT** operation

Step 1: Search for the key $k$ as if performing $\text{FIND}(k)$.

**SPLIT** 4-nodes as we go down

Step 2: If the leaf is a 2-node,

insert $(x, k)$, converting it into a 3-node

Step 3: If the leaf is a 3-node,

insert $(x, k)$, converting it into a 4-node

To perform $\text{INSERT}(x, k)$,
The **INSERT** operation

To perform **INSERT** \( (x, k) \),

**Step 1:** Search for the key \( k \) as if performing **FIND** \( (k) \).

**Split 4-nodes as we go down**

**Step 2:** If the leaf is a 2-node,

insert \( (x, k) \), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \( (x, k) \), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**(x, k),

**Step 1**: Search for the key k as if performing **FIND**(k).

**Step 2**: If the leaf is a 2-node,

insert (x, k), converting it into a 3-node

**Step 3**: If the leaf is a 3-node,

insert (x, k), converting it into a 4-node

**SPLIT** this!
The **INSERT** operation

To perform **INSERT**(\(x, k\)),

**Step 1**: Search for the key \(k\) as if performing **FIND**(\(k\)).

**SPLIT** 4-nodes as we go down

**Step 2**: If the leaf is a 2-node,
insert \((x, k)\), converting it into a 3-node

**Step 3**: If the leaf is a 3-node,
insert \((x, k)\), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**(\(x, k\)),

**Step 1:** Search for the key \(k\) as if performing **FIND**(\(k\)).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**(\(x, k\)),

**Step 1**: Search for the key \(k\) as if performing **FIND**(\(k\)).

**Step 2**: If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3**: If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**($x, k$),

**Step 1:** Search for the key $k$ as if performing **FIND**($k$).

**SPLIT** 4-nodes as we go down

**Step 2:** If the leaf is a 2-node,

insert ($x, k$), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert ($x, k$), converting it into a 4-node
To perform \textsc{insert}(x, k),

\textbf{Step 1:} Search for the key $k$ as if performing \textsc{find}(k).

\textbf{Step 2:} If the leaf is a 2-node,
insert $(x, k)$, converting it into a 3-node

\textbf{Step 3:} If the leaf is a 3-node,
insert $(x, k)$, converting it into a 4-node
The *INSERT* operation

To perform *INSERT*(\(x, k\)),

**Step 1:** Search for the key \(k\) as if performing *FIND*(\(k\)).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node

*SPLIT* 4-nodes as we go down

---

**Diagram:**

```
  2
  / \  \
 6   9
 / \ / \  \\
5   7 10 12
```

```
13 15
 / \  \\
14 16 17
```

```
24
 / \  \\
19 22
```

```
25 26
```

---

**Legend:**

- **INSERT**\((x, 8)\)
- **SPLIT** this!
The **INSERT** operation

To perform **INSERT** \((x, k)\),

**Step 1**: Search for the key \(k\) as if performing **FIND** \((k)\).

**Step 2**: If the leaf is a 2-node,
insert \((x, k)\), converting it into a 3-node

**Step 3**: If the leaf is a 3-node,
insert \((x, k)\), converting it into a 4-node
The `INSERT` operation

To perform `INSERT(x, k)`,

**Step 1:** Search for the key `k` as if performing `FIND(k)`.

**Step 2:** If the leaf is a 2-node,

insert `(x, k)`, converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert `(x, k)`, converting it into a 4-node

- **SPLIT** 4-nodes as we go down
The **INSERT** operation

To perform **INSERT**(x, k),

**Step 1:** Search for the key k as if performing **FIND**(k).

**Step 2:** If the leaf is a 2-node, insert (x, k), converting it into a 3-node

**Step 3:** If the leaf is a 3-node, insert (x, k), converting it into a 4-node

**SPLIT 4-nodes as we go down**
The **INSERT** operation

To perform **INSERT** \((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND** \((k)\).

**Step 2:** If the leaf is a 2-node,

insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert \((x, k)\), converting it into a 4-node
The **INSERT** operation

To perform **INSERT**(*x*, *k*),

**Step 1:** Search for the key *k* as if performing **FIND**(*k*).

**Step 2:** If the leaf is a 2-node,

- insert (**x**, *k*), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

- insert (**x**, *k*), converting it into a 4-node
To perform \( \text{INSERT}(x, k) \),

**Step 1:** Search for the key \( k \) as if performing \( \text{FIND}(k) \).

**Step 2:** If the leaf is a 2-node,
insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,
insert \((x, k)\), converting it into a 4-node

OK, one more thing...
The **INSERT** operation

To perform **INSERT**($x, k$),

**Step 1:** Search for the key $k$ as if performing **FIND**($k$).

**Step 2:** If the leaf is a 2-node, 
insert ($x, k$), converting it into a 3-node

**Step 3:** If the leaf is a 3-node, 
insert ($x, k$), converting it into a 4-node

OK, one more thing...
The **INSERT** operation

**Step 1:** Search for the key $k$ as if performing **FIND**($k$).

- **SPLIT 4-nodes as we go down**

**Step 2:** If the leaf is a 2-node,
- insert ($x, k$), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,
- insert ($x, k$), converting it into a 4-node

**OK, one more thing...**
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**Step 2:** If the leaf is a 2-node,

\[\text{insert } (x, k), \text{ converting it into a 3-node}\]

**Step 3:** If the leaf is a 3-node,

\[\text{insert } (x, k), \text{ converting it into a 4-node}\]

OK, one more thing... what happens when we **SPLIT** the root?
The **INSERT** operation

To perform **INSERT**\((x, k)\),

**Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).

**SPLIT** 4-nodes as we go down

**Step 2:** If the leaf is a 2-node, 
insert \((x, k)\), converting it into a 3-node

**Step 3:** If the leaf is a 3-node, 
insert \((x, k)\), converting it into a 4-node

OK, one more thing… what happens when we **SPLIT** the root?
The **INSERT** operation

To perform **INSERT**(*x*, *k*),

**Step 1:** Search for the key *k* as if performing **FIND**(*k*).

**SPLIT** 4-nodes as we go down

**Step 2:** If the leaf is a 2-node,

insert (*x*, *k*), converting it into a 3-node

**Step 3:** If the leaf is a 3-node,

insert (*x*, *k*), converting it into a 4-node

OK, one more thing... what happens when we **SPLIT** the root?
The **INSERT** operation

\[
\text{INSERT}(x, 20)
\]
The **INSERT** operation

![Tree Diagram]

**SPLITTING** the root increases the height of the tree and increases the length of all root-leaf paths by one.

So it maintains the **perfect balance** property:

- *i.e every path from the root to a leaf has the same length*
The **INSERT** operation

**INSERT**\((x, 20)\)

**SPLITTING** the root increases the height of the tree and increases the length of all root-leaf paths by one.

So it maintains the **perfect balance** property

- *i.e every path from the root to a leaf has the same length*

This is the only way **INSERT** can affect the length of paths so it also maintains the **perfect balance** property.
The **INSERT** operation

**INSERT**($x, 20$)

**Splitting** the root increases the height of the tree and increases the length of all root-leaf paths by one.

So it maintains the **perfect balance** property

- *i.e every path from the root to a leaf has the same length*

This is the only way **INSERT** can affect the length of paths so it also maintains the **perfect balance** property.

As each **SPLIT** takes $O(1)$ time, overall **INSERT** takes $O(\log n)$ time.
The **INSERT** operation

To perform **INSERT**\((x, k)\),

1. **Step 1:** Search for the key \(k\) as if performing **FIND**\((k)\).  

2. **Step 2:** If the bottom node is a 2-node, insert \((x, k)\), converting it into a 3-node.

3. **Step 3:** If the bottom node is a 3-node, insert \((x, k)\), converting it into a 4-node.

As each **SPLIT** takes \(O(1)\) time, overall **INSERT** takes \(O(\log n)\) time.
The **DELETE** operation

To perform **DELETE**\((k)\) on a leaf (we’ll deal with other nodes later)
The **DELETE** operation

To perform $\text{DELETE}(k)$ on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key $k$ using $\text{FIND}(k)$. 
The **DELETE** operation

To perform **DELETE**(\(k\)) **on a leaf** (we’ll deal with other nodes later)

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).
To perform \texttt{DELETE}(k) \textbf{on a leaf} (we'll deal with other nodes later)

\textbf{Step 1:} Search for the key $k$ using \texttt{FIND}(k).
To perform \texttt{DELETE}(k) on a leaf (we’ll deal with other nodes later)

\textbf{Step 1:} Search for the key \(k\) using \texttt{FIND}(k).
The **DELETE** operation

To perform **DELETE**(k) on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key k using **FIND**(k).
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf (**we’ll deal with other nodes later**)

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

**Step 2:** If the leaf is a 3-node,

- delete \((x, k)\), converting it into a 2-node

![Diagram of a B-tree with a DELETE operation on node 16 highlighted]
To perform **DELETE**\((k)\) on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key \(k\) using **FIND**\((k)\).

**Step 2:** If the leaf is a 3-node, delete \((x, k)\), converting it into a 2-node.
The **DELETE** operation

To perform **DELETE**($k$) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key $k$ using **FIND**($k$).
**Step 2:** If the leaf is a 3-node,
- delete ($x$, $k$), converting it into a 2-node
The **DELETE** operation

To perform **DELETE**(\(k\)) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

**Step 2:** If the leaf is a 3-node,

 delete \((x, k)\), converting it into a 2-node
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

**Step 2:** If the leaf is a 3-node,
   delete \((x, k)\), converting it into a 2-node
To perform **DELETE**($k$) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key $k$ using **FIND**($k$).

**Step 2:** If the leaf is a 3-node,

delete $(x, k)$, converting it into a 2-node.
The **DELETE** operation

To perform **DELETE**\((k)\) on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key **\(k\)** using **FIND**\((k)\).

**Step 2:** If the leaf is a 3-node,

delete \((x, k)\), converting it into a 2-node
The **DELETE** operation

To perform **DELETE**(*k*) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key *k* using **FIND**(*k*).

**Step 2:** If the leaf is a 3-node,
   delete (*x*, *k*), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete (*x*, *k*), converting it into a 3-node
To perform \texttt{DELETE}(k) on a leaf (we’ll deal with other nodes later)

\textbf{Step 1}: Search for the key \( k \) using \texttt{FIND}(k).

\textbf{Step 2}: If the leaf is a 3-node, delete \((x, k)\), converting it into a 2-node.

\textbf{Step 3}: If the leaf is a 4-node, delete \((x, k)\), converting it into a 3-node.
The **DELETE** operation

To perform **DELETE**($k$) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key $k$ using **FIND**($k$).

**Step 2:** If the leaf is a 3-node,
   delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete $(x, k)$, converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(\(k\)) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

**Step 2:** If the leaf is a 3-node,
   delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**\((k)\) on a leaf (**we’ll deal with other nodes later**)

**Step 1**: Search for the key \(k\) using **FIND**\((k)\).

**Step 2**: If the leaf is a 3-node,

- delete \((x, k)\), converting it into a 2-node

**Step 3**: If the leaf is a 4-node,

- delete \((x, k)\), converting it into a 3-node
The DELETE operation

To perform DELETE($k$) on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key $k$ using FIND($k$).

**Step 2:** If the leaf is a 3-node,
   delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete $(x, k)$, converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

**Step 2:** If the leaf is a 3-node,
   delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete \((x, k)\), converting it into a 3-node
To perform $\text{DELETE}(k)$ on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key $k$ using $\text{FIND}(k)$.

**Step 2:** If the leaf is a 3-node,
delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
delete $(x, k)$, converting it into a 3-node

**Step 4:** If the leaf is a 2-node,
To perform \texttt{DELETE}(k) on a leaf (we’ll deal with other nodes later)

\begin{itemize}
  \item \textbf{Step 1}: Search for the key $k$ using \texttt{FIND}(k).
  \item \textbf{Step 2}: If the leaf is a 3-node,
    \begin{itemize}
      \item delete $(x, k)$, converting it into a 2-node
    \end{itemize}
  \item \textbf{Step 3}: If the leaf is a 4-node,
    \begin{itemize}
      \item delete $(x, k)$, converting it into a 3-node
    \end{itemize}
  \item \textbf{Step 4}: If the leaf is a 2-node, ???
\end{itemize}
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

**Step 2:** If the leaf is a 3-node,

delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,

delete \((x, k)\), converting it into a 3-node

**Step 4:** If the leaf is a 2-node, ??? **We will make sure this never happens**
**Fusing 2-nodes**

We can fuse two 2-nodes (with the same parent) into a 4-node if that parent isn’t a 2-node.
Fusing 2-nodes

We can fuse two 2-nodes (with the same parent) into a 4-node if that parent isn't a 2-node.

BEFORE

AFTER
**Fusing 2-nodes**

We can **fuse** two 2-nodes (with the same parent) into a 4-node if that parent isn’t a 2-node.

**Before**

![Before diagram]

**After**

The extra key is pulled down from the parent (so it won’t work if the parent is a 2-node).
**Fusing 2-nodes**

We can fuse two 2-nodes (with the same parent) into a 4-node if that parent isn't a 2-node.

This is the opposite of a split operation.

**Before**

**After**

The extra key is pulled down from the parent (so it won’t work if the parent is a 2-node).
Fusing 2-nodes

We can fuse two 2-nodes (with the same parent) into a 4-node if that parent isn't a 2-node.

This is the opposite of a SPLIT operation.

Before

AFTER

The extra key is pulled down from the parent (so it won't work if the parent is a 2-node).

These subtrees haven't changed.
**Fusing 2-nodes**

We can fuse two 2-nodes (with the same parent) into a 4-node if that parent isn't a 2-node.

This is the opposite of a SPLIT operation.

**Before**

**After**

The extra key is pulled down from the parent (so it won’t work if the parent is a 2-node).

No path lengths have changed.

These subtrees haven’t changed.
FUSING 2-nodes

We can **Fuse** two 2-nodes (with the same parent) into a 4-node if that parent isn’t a 2-node.

This is the opposite of a **Split** operation.

**BEFORE**

The extra key is pulled down from the parent (so it won’t work if the parent is a 2-node).

no path lengths have changed (if it was perfectly balanced, it still is)

**AFTER**

these subtrees haven’t changed
**Fusing 2-nodes**

We can fuse two 2-nodes (with the same parent) into a 4-node if that parent isn't a 2-node.

This is the opposite of a split operation.

**BEFORE**

**AFTER**

The extra key is pulled down from the parent (so it won’t work if the parent is a 2-node).

No path lengths have changed.

*(if it was perfectly balanced, it still is)*

Fuse takes $O(1)$ time.
Transfering keys

If there is a 2-node and a 3-node (with the same parent), we can perform a Transfer (even if the parent is the root)
TRANSFERING keys

If there is a 2-node and a 3-node
(with the same parent)
we can perform a TRANSFER
(even if the parent is the root)
TRANSFERING keys

If there is a 2-node and a 3-node (with the same parent) we can perform a TRANSFER (even if the parent is the root)

BEFORE

AFTER

The keys have been rearranged
**TRANSFERING** keys

If there is a 2-node and a 3-node (with the same parent) we can perform a **TRANSFER** (even if the parent is the root)

**BEFORE**

**AFTER**

The keys have been rearranged

these subtrees haven’t changed
If there is a 2-node and a 3-node (with the same parent), we can perform a **Transfer** (even if the parent is the root).

The keys have been rearranged, no path lengths have changed.
TRANSFERRING keys

If there is a 2-node and a 3-node
(with the same parent)
we can perform a TRANSFER
(even if the parent is the root)

BEFORE

AFTER

The keys have been rearranged

no path lengths have changed
(if it was perfectly balanced, it still is)

these subtrees haven’t changed
If there is a 2-node and a 3-node (with the same parent)
we can perform a **Transfer** (even if the parent is the root)

*Transfer* takes $O(1)$ time

The keys have been rearranged

no path lengths have changed

(if it was perfectly balanced, it still is)

**Transfer** takes $O(1)$ time
Transfering keys

If there is a 2-node and a 3-node (with the same parent)
we can perform a Transfer (even if the parent is the root)

Transfer also works with a 2-node and a 4-node

The keys have been rearranged
no path lengths have changed
(if it was perfectly balanced, it still is)
Transfer takes $O(1)$ time
The **DELETE** operation

To perform $\text{DELETE}(k)$ on a leaf (we'll deal with other nodes later)

**Step 1:** Search for the key $k$ using $\text{FIND}(k)$.
   - use **Fuse** and **Transfer** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
   - delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   - delete $(x, k)$, converting it into a 3-node
The **DELETE** operation

To perform **DELETE**\( (k) \) on a leaf (we’ll deal with other nodes later)

**Step 1:** Search for the key \( k \) using **FIND**\( (k) \).

use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,

delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,

delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).
use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(k) on a leaf, *(we’ll deal with other nodes later)*

**Step 1:** Search for the key k using **FIND**(k).
  use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
  delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
  delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(*k*) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key *k* using **FIND**(*k*).
   
   use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
   
   delete (**x**, *k*), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   
   delete (**x**, *k*), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(*k*) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key *k* using **FIND**(*k*).

use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,

delete *(x, k)*, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,

delete *(x, k)*, converting it into a 3-node
The **DELETE** operation

**Step 1:** Search for the key $k$ using **FIND**( $k$).
   - use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
   - delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   - delete $(x, k)$, converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(k) on a leaf *(we'll deal with other nodes later)*

**Step 1:** Search for the key *k* using **FIND**(k).
use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
delete *(x, k)*, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
delete *(x, k)*, converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf (*we’ll deal with other nodes later*)

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,

delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,

delete \((x, k)\), converting it into a 3-node
To perform **DELETE**(\(k\)) on a leaf (we'll deal with other nodes later)

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).
use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(k) **on a leaf** *(we’ll deal with other nodes later)*

**Step 1:** Search for the key k using **FIND**(k).
use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
delete (x, k), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
delete (x, k), converting it into a 3-node
The **DELETE** operation

**Step 1:** Search for the key $k$ using $\text{FIND}(k)$.

use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,

delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,

delete $(x, k)$, converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(\(k\)) on a leaf *(we'll deal with other nodes later)*

**Step 1**: Search for the key \(k\) using **FIND**(\(k\)).

- use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2**: If the leaf is a 3-node,
- delete \((x, k)\), converting it into a 2-node

**Step 3**: If the leaf is a 4-node,
- delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

Now we can **DELETE** the 5

To perform **DELETE**(*k*) on a leaf (*we’ll deal with other nodes later*)

**Step 1:** Search for the key *k* using **FIND**(*k*).  
use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,  
delete (x, *k*), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,  
delete (x, *k*), converting it into a 3-node
The **DELETE** operation

Now we can **DELETE** the 5

To perform **DELETE**(5) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \( k \) using **FIND**(\( k \)).  
use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,  
delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,  
delete \((x, k)\), converting it into a 3-node
The **DELETE** operation

To perform **DELETE**(\(k\)) **on a leaf** *(we'll deal with other nodes later)*

**Step 1:** Search for the key \(k\) using **FIND**(\(k\)).

use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,

delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,

delete \((x, k)\), converting it into a 3-node
The DELETE operation

To perform DELETE($k$) on a leaf (we'll deal with other nodes later)

**Step 1:** Search for the key $k$ using FIND($k$).
   use FUSE and TRANSFER to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
   delete $(x, k)$, converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete $(x, k)$, converting it into a 3-node

OK, one more thing...
The **DELETE** operation

To perform **DELETE**(k) on a leaf *(we’ll deal with other nodes later)*

**Step 1:** Search for the key \( k \) using **FIND**(k).
   use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2:** If the leaf is a 3-node,
   delete \((x, k)\), converting it into a 2-node

**Step 3:** If the leaf is a 4-node,
   delete \((x, k)\), converting it into a 3-node

OK, one more thing… what happens when we FUSE the root?
**Fusing** the root

We said that we could only **fuse** two 2-nodes if the parent was not a 2-node... we make an exception for the root.

**Fusing** the root can decrease the height of the tree, which in turn decreases the length of all root-leaf paths by one
**FUSING** the root

We said that we could only **Fuse** two 2-nodes if the parent was not a 2-node... we make an exception for the root

![Diagram](image)

**FUSING** the root can decrease the height of the tree which in turn decreases the length of all root-leaf paths by one

So it maintains the **perfect balance** property

- *i.e every path from the root to a leaf has the same length*
**Fusing** the root

We said that we could only **Fuse** two 2-nodes if the parent was not a 2-node... we make an exception for the root

Fusing the root can decrease the height of the tree which in turn decreases the length of all root-leaf paths by one

So it maintains the **perfect balance** property

- *i.e every path from the root to a leaf has the same length*

This is the only way **Delete** can affect the length of paths so it also maintains the **perfect balance** property
**Fusing** the root

We said that we could only **Fuse** two 2-nodes if the parent was not a 2-node... we make an exception for the root.

![Tree Diagram]

**Fusing** the root can decrease the height of the tree which in turn decreases the length of all root-leaf paths by one.

So it maintains the **perfect balance** property:
- i.e every path from the root to a leaf has the same length.

This is the only way **Delete** can affect the length of paths so it also maintains the **perfect balance** property.

As each **Fuse** or **Transfer** takes \( O(1) \) time, overall **Delete** takes \( O(\log n) \) time.
The **DELETE** operation

To perform **DELETE**(k) on a leaf *(we’ll deal with other nodes later)*

**Step 1**: Search for the key k using **FIND**(k).
   use **FUSE** and **TRANSFER** to convert 2-nodes as we go down

**Step 2**: If the leaf is a 3-node,
   delete (x, k), converting it into a 2-node

**Step 3**: If the leaf is a 4-node,
   delete (x, k), converting it into a 3-node

As each **FUSE** or **TRANSFER** takes \(O(1)\) time, overall **DELETE** takes \(O(\log n)\) time
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?
What if we want to DELETE something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of $k$ (this is essentially the same as **FIND**)
- that’s the element with the largest key $k'$ such that $k' < k$
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of $k$ (this is essentially the same as **FIND**)
- that’s the element with the largest key $k'$ such that $k' < k$
What if we want to DELETE something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \(k\) (this is essentially the same as **FIND**)
- that’s the element with the largest key \(k'\) such that \(k' < k\)
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of $k$ (this is essentially the same as **FIND**)
- that’s the element with the largest key $k'$ such that $k' < k$
What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)
- that’s the element with the largest key \( k' \) such that \( k' < k \)
What if we want to DELETE something other than a leaf?

**Step 1:** Find the **Predecessor** of \( k \) (this is essentially the same as **Find**)
- that’s the element with the largest key \( k' \)
  such that \( k' < k \)
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)
- that’s the element with the largest key \( k' \)
  such that \( k' < k \)

**Step 2:** Call **DELETE**\((k')\)
- fortunately \( k' \) is always a leaf
What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of $k$ (this is essentially the same as **FIND**)
- that’s the element with the largest key $k'$ such that $k' < k$

**Step 2:** Call **DELETE($k'$)**
- fortunately $k'$ is *always* a leaf
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of $k$ (this is essentially the same as **FIND**)  
- that’s the element with the largest key $k'$ such that $k' < k$

**Step 2:** Call **DELETE**($k'$)  
- fortunately $k'$ is *always* a leaf

**Step 3:** Overwrite $k$ with another copy of $k'$
What if we want to DELETE something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)
- that’s the element with the largest key \( k' \) such that \( k' < k \)

**Step 2:** Call **DELETE** \( (k') \)
- fortunately \( k' \) is always a leaf

**Step 3:** Overwrite \( k \) with another copy of \( k' \)
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)  
- that’s the element with the largest key \( k' \) such that \( k' < k \)

**Step 2:** Call **DELETE**(\( k' \))  
- fortunately \( k' \) is *always* a leaf

**Step 3:** Overwrite \( k \) with another copy of \( k' \)
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)
- that’s the element with the largest key \( k' \) such that \( k' < k \)

**Step 2:** Call **DELETE**\((k')\)
- fortunately \( k' \) is always a leaf

**Step 3:** Overwrite \( k \) with another copy of \( k' \)
What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)
- that’s the element with the largest key \( k' \)
  such that \( k' < k \)

**Step 2:** Call **DELETE**\((k')\)
- fortunately \( k' \) is *always* a leaf

**Step 3:** Overwrite \( k \) with another copy of \( k' \)
The **DELETE** operation

What if we want to **DELETE** something other than a leaf?

**Step 1:** Find the **PREDECESSOR** of \( k \) (this is essentially the same as **FIND**)  
- that’s the element with the largest key \( k' \) such that \( k' < k \)

**Step 2:** Call **DELETE**\((k')\)  
- fortunately \( k' \) is always a leaf

**Step 3:** Overwrite \( k \) with another copy of \( k' \)  
This also takes \( O(\log n) \) time
2-3-4 tree summary

A 2-3-4 is a data structure based on a tree structure which supports \texttt{INSERT}(x, k), \texttt{FIND}(k) and \texttt{DELETE}(k)

each of these operations takes \textit{worst case} $O(\log n)$ time
2-3-4 tree summary

A 2-3-4 is a data structure based on a tree structure which supports \textsc{Insert}(x, k), \textsc{Find}(k) and \textsc{Delete}(k)

each of these operations takes \textit{worst case} $O(\log n)$ time

\textit{Unfortunately, 2-3-4 trees are awkward to implement because the nodes don’t all have the same number of children}
2-3-4 tree summary

A 2-3-4 is a data structure based on a tree structure which supports \textbf{INSERT}(x, k), \textbf{FIND}(k) and \textbf{DELETE}(k)

each of these operations takes \textit{worst case} $O(\log n)$ time

\textit{Unfortunately, 2-3-4 trees are awkward to implement because the nodes don’t all have the same number of children}

So, what is used in practice?
Red-Black tree summary

A Red-Black tree is a data structure based on a **binary** tree structure which supports \texttt{INSERT}(x, k), \texttt{FIND}(k) and \texttt{DELETE}(k)

which of these operations takes \textit{worst case} $O(\log n)$ time
Red-Black tree summary

A Red-Black tree is a data structure based on a **binary** tree structure which supports **INSERT**(x, k), **FIND**(k) and **DELETE**(k)

The root is **black**
Red-Black tree summary

A Red-Black tree is a data structure based on a binary tree structure which supports \text{INSERT}(x, k), \text{FIND}(k) and \text{DELETE}(k)

each of these operations takes \textit{worst case} $O(\log n)$ time

The root is \textbf{black}

All root-to-leaf paths have the same number of \textbf{black} nodes
Red-Black tree summary

A Red-Black tree is a data structure based on a binary tree structure which supports \textsc{Insert}(x, k), \textsc{Find}(k) and \textsc{Delete}(k)

The root is black

All root-to-leaf paths have the same number of black nodes

Red nodes cannot have red children

Each of these operations takes worst case $O(\log n)$ time
Red-Black tree summary

A Red-Black tree is a data structure based on a binary tree structure which supports \textbf{INSERT}(x, k), \textbf{FIND}(k) and \textbf{DELETE}(k) each of these operations takes \textit{worst case} $O(\log n)$ time.

The root is \textbf{black}

All root-to-leaf paths have the same number of \textbf{black} nodes

\textbf{Red} nodes cannot have \textbf{red} children

\textit{If these are used in practice, why did you waste our time with 2-3-4 trees?}
Red-Black tree summary

A Red-Black tree is a data structure based on a binary tree structure which supports \textsc{Insert}(x, k), \textsc{Find}(k) and \textsc{Delete}(k) each of these operations takes worst case $O(\log n)$ time.

![Red-Black Tree Diagram]

The root is \textbf{black}

- All root-to-leaf paths have the same number of \textbf{black} nodes
- \textbf{Red} nodes cannot have \textbf{red} children

\textit{If these are used in practice, why did you waste our time with 2-3-4 trees?}

1. 2-3-4 trees are conceptually much nicer
A Red-Black tree is a data structure based on a binary tree structure which supports $\text{INSERT}(x, k)$, $\text{FIND}(k)$ and $\text{DELETE}(k)$ each of these operations takes worst case $O(\log n)$ time

The root is black
All root-to-leaf paths have the same number of black nodes
Red nodes cannot have red children

If these are used in practice, why did you waste our time with 2-3-4 trees?
1. 2-3-4 trees are conceptually much nicer  2. they are secretly the same :}
2-3-4 trees vs. Red-Black trees

Any 2-3-4 tree can be converted into a Red-Black tree (and visa-versa)

The operations on 2-3-4 trees also have equivalent operations on a Red-Black tree
(the details of the Red-Black tree operations are in CLRS Chapter 13)
2-3-4 trees vs. Red-Black trees

Any 2-3-4 tree can be converted into a Red-Black tree (and visa-versa)

The operations on 2-3-4 trees also have equivalent operations on a Red-Black tree

(the details of the Red-Black tree operations are in CLRS Chapter 13)
2-3-4 trees vs. Red-Black trees

Any 2-3-4 tree can be converted into a Red-Black tree (and visa-versa)

The operations on 2-3-4 trees also have equivalent operations on a Red-Black tree

(the details of the Red-Black tree operations are in CLRS Chapter 13)
2-3-4 trees vs. Red-Black trees

Any 2-3-4 tree can be converted into a Red-Black tree (and visa-versa)

The operations on 2-3-4 trees also have equivalent operations on a Red-Black tree

(the details of the Red-Black tree operations are in CLRS Chapter 13)
2-3-4 trees vs. Red-Black trees

Any 2-3-4 tree can be converted into a Red-Black tree (and visa-versa)

The operations on 2-3-4 trees also have equivalent operations on a Red-Black tree
(the details of the Red-Black tree operations are in CLRS Chapter 13)
2-3-4 trees vs. Red-Black trees

Any 2-3-4 tree can be converted into a Red-Black tree (and visa-versa)

The operations on 2-3-4 trees also have equivalent operations on a Red-Black tree

(the details of the Red-Black tree operations are in CLRS Chapter 13)
Dynamic Search Structure Summary

A **dynamic search structure** supports (at least) the following three operations:

- **DELETE**\( (k) \) - deletes the (unique) element \( x \) with \( x.key = k \)
- **INSERT**\( (x, k) \) - inserts \( x \) with key \( k = x.key \)
- **FIND**\( (k) \) - returns the (unique) element \( x \) with \( x.key = k \)

Here are the *worst case* time complexities of the structures we have seen…

<table>
<thead>
<tr>
<th></th>
<th>INSERT</th>
<th>DELETE</th>
<th>FIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>( O(1) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>2-3-4 Tree</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Red-Black Tree</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>
End of part one
Part two
Skip lists

inspired by slides by Ashley Montanaro
Dynamic Search Structures

A dynamic search structure, stores a set of elements

Each element \( x \) must have a unique key - \( x\.key \)

The following operations are supported:

- \( \text{INSERT}(x, k) \) - inserts \( x \) with key \( k = x\.key \)
- \( \text{FIND}(k) \) - returns the (unique) element \( x \) with \( x\.key = k \).
  
  (or reports that it doesn't exist)
- \( \text{DELETE}(k) \) - deletes the (unique) element \( x \) with \( x\.key = k \).
  
  (or reports that it doesn't exist)

We would also like it to support:

- \( \text{PREDECESSOR}(k) \) - returns the (unique) element \( x \)
  
  with the largest key such that \( x\.key < k \)
- \( \text{RANGEFIND}(k_1, k_2) \) - returns every element \( x \) with \( k_1 \leq x\.key \leq k_2 \)
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

![Sorted Linked List Diagram]

The bottleneck is **FIND**, which is very inefficient,
- we have to look through the entire linked list to find an item (in the worst case)

**INSERT** and **DELETE** also take $O(n)$ time but only because they rely on **FIND**

How can we speed up the **FIND** operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

The bottleneck is **FIND**, which is *very inefficient*,
- we have to look through the entire linked list to find an item *(in the worst case)*

**INSERT** and **DELETE** also take $O(n)$ time *but only because they rely on FIND*

How can we speed up the **FIND** operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

The bottleneck is **FIND**, which is very inefficient,
- we have to look through the entire linked list to find an item (in the worst case)

**INSERT** and **DELETE** also take $O(n)$ time but only because they rely on **FIND**

How can we speed up the **FIND** operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

The bottleneck is `FIND`, which is very inefficient,
- we have to look through the entire linked list to find an item (in the worst case)

`INSERT` and `DELETE` also take $O(n)$ time but only because they rely on `FIND`

How can we speed up the `FIND` operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

The bottleneck is `FIND`, which is very inefficient,
- we have to look through the entire linked list to find an item (in the worst case)

`INSERT` and `DELETE` also take $O(n)$ time but only because they rely on `FIND`

How can we speed up the `FIND` operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a *sorted* Linked List?

The bottleneck is \texttt{FIND}, which is very inefficient,
- we have to look through the entire linked list to find an item \textit{(in the worst case)}.

\texttt{INSERT} and \texttt{DELETE} also take \(O(n)\) time \textit{but only because they rely on FIND.}

How can we speed up the \texttt{FIND} operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

The bottleneck is **FIND**, which is *very inefficient*,
- we have to look through the entire linked list to find an item *(in the worst case)*

**INSERT** and **DELETE** also take \( O(n) \) time *but only because they rely on **FIND***

How can we speed up the **FIND** operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

The bottleneck is \texttt{FIND}, which is \textit{very inefficient},

- we have to look through the entire linked list to find an item \texttt{(in the worst case)}

\texttt{INSERT} and \texttt{DELETE} also take \(O(n)\) time \textit{but only because they rely on \texttt{FIND}}

How can we speed up the \texttt{FIND} operation?
Using a Linked List as a Dynamic Search Structure (again)

Earlier we briefly considered using an unsorted Linked List as a dynamic search structure.

What about using a sorted Linked List?

![Linked List Diagram]

The bottleneck is FIND, which is very inefficient, - we have to look through the entire linked list to find an item (in the worst case).

INSERT and DELETE also take $O(n)$ time but only because they rely on FIND.

How can we speed up the FIND operation?
Making Shortcuts

How about adding some shortcuts?
Making Shortcuts

How about adding some shortcuts?

1 → 2 → 5 → 9 → 16 → 18 → 25
Making Shortcuts

How about adding some shortcuts?
Making Shortcuts

How about adding some shortcuts?

1  2  5  9  16  18  25
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys . . .

To perform \textsc{Find}(k) we start in the top list
and go right until we come to a key $k' > k$
then we move down to the bottom list
and go right until we find $k$
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...

To perform \texttt{FIND}(k) we start in the top list and go right until we come to a key $k' > k$

then we move down to the bottom list and go right until we find $k$
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...

To perform \texttt{FIND}(k) we start in the top list
and go right until we come to a key $k' > k$
then we move down to the bottom list
and go right until we find $k$
How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...

To perform $\text{FIND}(k)$ we start in the top list and go right until we come to a key $k' > k$.

then we move down to the bottom list and go right until we find $k$. 
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys... 

To perform $\text{FIND}(k)$ we start in the top list

and go right until we come to a key $k' > k$

then we move down to the bottom list

and go right until we find $k$
How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...

To perform $\text{FIND}(k)$ we start in the top list and go right until we come to a key $k' > k$.

then we move down to the bottom list and go right until we find $k$.
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys... 

To perform \texttt{FIND}(k) we start in the top list and go right until we come to a key $k' > k$ then we move down to the bottom list and go right until we find $k$
How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...

To perform $\text{FIND}(k)$ we start in the top list
and go right until we come to a key $k' > k$
then we move down to the bottom list
and go right until we find $k$
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys... 

To perform \text{FIND}(k) we start in the top list and go right until we come to a key \( k' > k \)

then we move down to the bottom list and go right until we find \( k \).
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys...

To perform $\text{FIND}(k)$ we start in the top list
and go right until we come to a key $k' > k$
then we move down to the bottom list
and go right until we find $k$

How long does this take?
Making Shortcuts

How about adding some shortcuts?

We’ve attached a second linked list containing only some of the keys . . .

To perform \texttt{FIND}(k) we start in the top list and go right until we come to a key \( k' > k \)

then we move down to the bottom list and go right until we find \( k \)

How long does this take?

That depends on where we place the shortcuts
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list...  
(the bottom list always contains all $n$ keys)
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list... *(the bottom list always contains all $n$ keys)*

where should we put them to minimise
the *worst case* time for a FIND operation?
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list. . .

*(the bottom list always contains all $n$ keys)*

where should we put them to minimise

the *worst case* time for a *Find* operation?
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list... 

*(the bottom list always contains all $n$ keys)*

where should we put them to minimise

the *worst case* time for a *Find* operation?

finding this is quick

finding this is slow
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list... 

*(the bottom list always contains all $n$ keys)*

where should we put them to minimise 

the *worst case* time for a FIND operation?
Linked Lists with two levels

Imagine that we decide to place \( m \) keys in the top list... (the bottom list always contains all \( n \) keys)

where should we put them to minimise

the \textit{worst case} time for a \texttt{FIND} operation?
Linked Lists with two levels

Imagine that we decide to place \( m \) keys in the top list... (the bottom list always contains all \( n \) keys)

where should we put them to minimise the \textit{worst case} time for a \texttt{FIND} operation?

![Diagram of linked lists with two levels]

finding this is slow

finding this is quick
Linked Lists with two levels

Imagine that we decide to place \( m \) keys in the top list...

*(the bottom list always contains all \( n \) keys)*

where should we put them to minimise

the *worst case* time for a **FIND** operation?
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list... (the bottom list always contains all $n$ keys)

where should we put them to minimise
the worst case time for a FIND operation?
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list... 
\((the \ bottom \ list \ always \ contains \ all \ n \ keys)\)

where should we put them to minimise 
the \textit{worst case} time for a \texttt{FIND} operation?

If we spread out the $m$ keys in the top list evenly ...
Imagine that we decide to place $m$ keys in the top list. . .

*(the bottom list always contains all $n$ keys)*

where should we put them to minimise

the *worst case* time for a *FIND* operation?

If we spread out the $m$ keys in the top list evenly . . .

the *worst case* time for a *FIND* operation becomes $O(m + n/m)$
Linked Lists with two levels

Imagine that we decide to place \( m \) keys in the top list . . .

\( \text{(the bottom list always contains all } n \text{ keys)} \)

where should we put them to minimise

the \textit{worst case} time for a \texttt{FIND} operation?

If we spread out the \( m \) keys in the top list evenly . . .

the \textit{worst case} time for a \texttt{FIND} operation becomes \( O(m + n/m) \)
Linked Lists with two levels

Imagine that we decide to place \( m \) keys in the top list . . .

\[ (\text{the bottom list always contains all } n \text{ keys}) \]

where should we put them to minimise

the \textbf{worst case} time for a \texttt{FIND} operation?

\[
\begin{align*}
\text{m} & \\
\approx \frac{n}{m} & \approx \frac{n}{m} & \approx \frac{n}{m}
\end{align*}
\]

If we spread out the \( m \) keys in the top list evenly . . .

the \textbf{worst case} time for a \texttt{FIND} operation becomes \( O(m + n/m) \)
Linked Lists with two levels

Imagine that we decide to place $m$ keys in the top list... (the bottom list always contains all $n$ keys)

where should we put them to minimise

the worst case time for a FIND operation?

If we spread out the $m$ keys in the top list evenly... 

the worst case time for a FIND operation becomes $O(m + n/m)$

By setting $m = \sqrt{n}$, we get

the worst case time for a FIND operation is $O(\sqrt{n})$
Linked Lists with many levels

How about adding even more lists? \( (each \ list \ is \ called \ a \ level) \)
Linked Lists with many levels

How about adding even more lists? (each list is called a level)

Each level will now contain half of the keys (rounding up) from the level below. They are chosen to be as evenly spread as possible.
Linked Lists with many levels

How about adding even more lists? (each list is called a level)

Each level will now contain half of the keys (rounding up) from the level below.

They are chosen to be as evenly spread as possible.
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each **level** will now contain **half** of the keys *(rounding up)* from the level below

They are chosen to be as evenly spread as possible
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each *level* will now contain **half** of the keys *(rounding up)* from the level below

They are chosen to be as evenly spread as possible
Linked Lists with many levels

How about adding even more lists? (each list is called a level)

Each level will now contain half of the keys (rounding up) from the level below

They are chosen to be as evenly spread as possible
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each level will now contain **half** of the keys *(rounding up)* from the level below

They are chosen to be as evenly spread as possible

---

Diagram:

```
1
  ↓
1
  ↓
1
  ↓
1

1
  ↓
5
  ↓
2
  ↓
5

16
  ↓
16
  ↓
16
  ↓
16
```

---

Numbers:

1, 2, 9, 16, 18, 25, 27, 31, 35, 38
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each **level** will now contain **half** of the keys *(rounding up)* from the level below

They are chosen to be as evenly spread as possible
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each **level** will now contain **half** of the keys *(rounding up)* from the level below

They are chosen to be as evenly spread as possible
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each level will now contain **half** of the keys *(rounding up)* from the level below.

They are chosen to be as evenly spread as possible.

The bottom level contains every key and every level contains the leftmost and rightmost keys.
Linked Lists with many levels

How about adding even more lists? *(each list is called a level)*

Each level will now contain **half** of the keys *(rounding up)* from the level below

They are chosen to be as evenly spread as possible

The bottom level contains every key and every level contains the leftmost and rightmost keys

As each level contains half of the keys from the level below, there are \( O(\log n) \) levels
**FIND** in multi-level linked lists

How do we perform **FIND**\((k)\) in multi-level linked list? 

*(essentially just like before)*
How do we perform \texttt{FIND}(k) in multi-level linked list?

(essentially just like before)

To perform \texttt{FIND}(k),

\begin{itemize}
  \item Start at the top-left (the head of the top level)
  \item While you haven’t found \( k \):
    \begin{itemize}
      \item If the node to the right’s key, \( k' \leq k \)
        \begin{itemize}
          \item Move right
        \end{itemize}
      \item Else
        \begin{itemize}
          \item Move down
        \end{itemize}
    \end{itemize}
\end{itemize}
**FIND** in multi-level linked lists

How do we perform **FIND**(\(k\)) in multi-level linked list?

(essentially just like before)

Start at the top-left (the head of the top level)

To perform **FIND**(\(k\)),

While you haven’t found \(k\):

If the node to the right’s key, \(k' \leq k\)

Move right

Else

Move down
How do we perform $\text{FIND}(k)$ in multi-level linked list?

(essentially just like before)

Start at the top-left (the head of the top level)

To perform $\text{FIND}(k)$,

While you haven’t found $k$:

If the node to the right’s key, $k' \leq k$

Move right

Else

Move down
**FIND** in multi-level linked lists

How do we perform \( \text{FIND}(k) \) in multi-level linked list?

(essentially just like before)

Start at the top-left (the head of the top level)

To perform \( \text{FIND}(k) \),

While you haven’t found \( k \):
  
  If the node to the right’s key, \( k' \leq k \)
    
    Move right
  
  Else
    
    Move down

consider \( \text{FIND}(35) \)
**FIND** in multi-level linked lists

How do we perform **FIND**(k) in multi-level linked list?

*(essentially just like before)*

To perform **FIND**(k),

Start at the top-left *(the head of the top level)*

While you haven’t found k:

- If the node to the right’s key, k’ ≤ k
  - Move right
- Else
  - Move down

Consider **FIND**(35)
**FIND** in multi-level linked lists

How do we perform FIND\((k)\) in multi-level linked list?

*(essentially just like before)*

Start at the top-left (the head of the top level)

To perform **FIND**\((k)\),

While you haven’t found \(k\):

- If the node to the right’s key, \(k' \leq k\)  
  Move right
- Else  
  Move down

Consider **FIND**\((35)\)
How do we perform \texttt{FIND}(k) in multi-level linked list?

(essentially just like before)

Start at the top-left (the head of the top level)

To perform \texttt{FIND}(k),

While you haven’t found \(k\):

If the node to the right’s key, \(k' \leq k\)

Move right

Else

Move down
**FIND** in multi-level linked lists

How do we perform $\text{FIND}(k)$ in multi-level linked list? (essentially just like before)

To perform $\text{FIND}(k)$,

- Start at the top-left (the head of the top level)
- While you haven’t found $k$:
  - If the node to the right’s key, $k' \leq k$:
    - Move right
  - Else:
    - Move down

Consider $\text{FIND}(35)$:

- $38 > 35$: cross out
- $16 \leq 35$:
- $38 > 35$: cross out
- $38 > 35$: cross out
- $38 > 35$: cross out
- $38 > 35$: cross out
- $38 > 35$: cross out
- $38 > 35$: cross out
**FIND in multi-level linked lists**

How do we perform $\text{FIND}(k)$ in multi-level linked list?

(essentially just like before)

Start at the top-left (the head of the top level)

To perform $\text{FIND}(k)$,

While you haven’t found $k$:

If the node to the right’s key, $k' \leq k$
   Move right

Else
   Move down
How do we perform \texttt{FIND}(k) in multi-level linked list? (essentially just like before)

To perform \texttt{FIND}(k),

Start at the top-left (the head of the top level)

While you haven’t found $k$:

- If the node to the right’s key, $k’ \leq k$
  Move right

- Else
  Move down
**FIND in multi-level linked lists**

How do we perform $FIND(k)$ in multi-level linked list?

*(essentially just like before)*

To perform $FIND(k)$,

**Algorithm**

1. Start at the top-left (*the head of the top level*)
2. While you haven’t found $k$:
   - If the node to the right’s key, $k' \leq k$
     - Move right
   - Else
     - Move down

Consider $FIND(35)$:

- $38 > 35$ (Move right)
- $16 \leq 35$ (Move right)
- $38 > 35$ (Move right)
- $25 \leq 35$ (Move right)
- $31 \leq 35$ (Move right)
- $38 > 35$ (Move right)

Result: $38$
**FIND** in multi-level linked lists

How do we perform \( \text{FIND}(k) \) in multi-level linked lists?

(essentially just like before)

To perform \( \text{FIND}(k) \),

Start at the top-left (the head of the top level)

While you haven’t found \( k \): 

If the node to the right’s key, \( k' \leq k \)

Move right

Else

Move down
How do we perform \texttt{FIND}(k) in multi-level linked list?

(essentially just like before)

Start at the top-left (the head of the top level)

To perform \texttt{FIND}(k),

While you haven’t found $k$:

If the node to the right’s key, $k' \leq k$

Move right

Else

Move down

$35 = 35!$
The complexity of \textsc{Find}

How long does $\textsc{Find}(k)$ take in a multi-level linked list?
The complexity of $\text{FIND}$

How long does $\text{FIND}(k)$ take in a multi-level linked list?

**Observation 1** We only move down at most $O(\log n)$ times because there are only $O(\log n)$ levels.
The complexity of \textsc{Find}

How long does \textsc{Find}(k) take in a multi-level linked list?

\textbf{Observation 1} We only move down at most $O(\log n)$ times

because there are only $O(\log n)$ levels

\textbf{Observation 2} Between any two nodes on level $i$,

there are at most 2 nodes on level $i + 1$

because we took half the nodes and spread them evenly
The complexity of FIND

How long does FIND($k$) take in a multi-level linked list?

**Observation 1** We only move down at most $O(\log n)$ times

because there are only $O(\log n)$ levels

**Observation 2** Between any two nodes on level $i$,
there are at most 2 nodes on level $i + 1$

because we took half the nodes and spread them evenly

**Observation 3** We only move right at most 2 times on any level $i + 1$

because we stopped moving right on level $i$
The complexity of **FIND**

How long does **FIND**($k$) take in a multi-level linked list?

**Observation 1** We only move down at most $O(\log n)$ times 

*because there are only $O(\log n)$ levels*

**Observation 2** Between any two nodes on level $i$, there are at most 2 nodes on level $i + 1$ 

*because we took half the nodes and spread them evenly*

![Diagram](image)

**Observation 3** We only move right at most 2 times on any level $i + 1$ 

*because we stopped moving right on level $i$*
The complexity of \textbf{FIND}

How long does \texttt{FIND}(k) take in a multi-level linked list?

\textbf{Observation 1} We only \textbf{move down} at most $O(\log n)$ times

\textit{because there are only $O(\log n)$ levels}

\textbf{Observation 2} Between any two nodes on level $i$,
\textit{there are at most 2 nodes on level $i + 1$}

\textit{because we took half the nodes and spread them evenly}

\textbf{Observation 3} We only \textbf{move right} at most 2 times on any level $i + 1$

\textit{because we stopped \textbf{moving right} on level $i$}
The complexity of \textit{FIND}

How long does \textit{FIND}(k) take in a multi-level linked list?

\textbf{Observation 1} We only \textbf{move down} at most $O(\log n)$ times
because there are only $O(\log n)$ levels.

\textbf{Observation 2} Between any two nodes on level $i$,
there are at most 2 nodes on level $i + 1$
because we took half the nodes and spread them evenly.

\textbf{Observation 3} We only \textbf{move right} at most 2 times on any level $i + 1$
because we stopped \textbf{moving right} on level $i$. 
The complexity of \textit{FIND}

How long does \text{FIND}(k) take in a multi-level linked list?

\textbf{Observation 1} We only move down at most \(O(\log n)\) times

\textit{because there are only} \(O(\log n)\) \textit{levels}

\textbf{Observation 2} Between any two nodes on level \(i\),

\textit{there are at most} 2 \textit{nodes on level} \(i + 1\)

\textit{because we took half the nodes and spread them evenly}

\textbf{Observation 3} We only move right at most 2 times on any level \(i + 1\)

\textit{because we stopped moving right on level} \(i\)
The complexity of \textsc{Find}

How long does $\textsc{find}(k)$ take in a multi-level linked list?

\textbf{Observation 1} We only move \textbf{down} at most $O(\log n)$ times \\
\textit{because there are only $O(\log n)$ levels}

\textbf{Observation 2} Between any two nodes on level $i$, \\
there are at most 2 nodes on level $i + 1$ \\
\textit{because we took half the nodes and spread them evenly}

\textbf{Observation 3} We only move \textbf{right} at most 2 times on any level $i + 1$ \\
\textit{because we stopped moving right on level $i$}
The complexity of **FIND**

How long does **FIND**(\(k\)) take in a multi-level linked list?

**Observation 1** We only **move down** at most \(O(\log n)\) times

*because there are only \(O(\log n)\) levels*

**Observation 2** Between any two nodes on level \(i\), there are at most 2 nodes on level \(i + 1\)

*because we took half the nodes and spread them evenly*

**Observation 3** We only **move right** at most 2 times on any level \(i + 1\)

*because we stopped **moving right** on level \(i\)*
The complexity of \textbf{FIND}.

How long does \texttt{FIND}(k) take in a multi-level linked list?

**Observation 1** We only move down at most $O(\log n)$ times

because there are only $O(\log n)$ levels.

**Observation 2** Between any two nodes on level $i$, there are at most 2 nodes on level $i + 1$

because we took half the nodes and spread them evenly.

**Observation 3** We only move right at most 2 times on any level $i + 1$

because we stopped moving right on level $i$.

**Fact** We only move at most $O(\log n)$ times while performing a \texttt{FIND}. 

Multi-level Linked Lists

If we had a multi-level linked list with $O(\log n)$ levels

where each level contained half of the keys from the level below

and the keys were evenly spread as possible

then we could perform FIND in $O(\log n)$ time
Multi-level Linked Lists

If we had a multi-level linked list with $O(\log n)$ levels

where each level contained half of the keys from the level below

and the keys were evenly spread as possible

then we could perform FIND in $O(\log n)$ time

How are we going to do INSERTS and DELETES?
Multi-level Linked Lists

If we had a multi-level linked list with $O(\log n)$ levels

where each level contained half of the keys from the level below

and the keys were evenly spread as possible

then we could perform FIND in $O(\log n)$ time

How are we going to do INSERTS and DELETES?

Which levels should we put an INSERTED key into?
Multi-level Linked Lists

If we had a multi-level linked list with $O(\log n)$ levels
where each level contained half of the keys from the level below
and the keys were evenly spread as possible
then we could perform FIND in $O(\log n)$ time

How are we going to do INSERTS and Deletes?
Which levels should we put an INSERTED key into?
How can we keep a good spread of keys at each levels?
Multi-level Linked Lists

If we had a multi-level linked list with \( O(\log n) \) levels

where each level contained half of the keys from the level below

and the keys were evenly spread as possible

then we could perform FIND in \( O(\log n) \) time

How are we going to do INSERTS and DELETES?

Which levels should we put an INSERTED key into?

How can we keep a good spread of keys at each levels?

especially when we don’t know what will be INSERTED and DELETED in the future
Multi-level Linked Lists

If we had a multi-level linked list with $O(\log n)$ levels

where each level contained half of the keys from the level below

and the keys were evenly spread as possible

then we could perform FIND in $O(\log n)$ time

How are we going to do INSERTS and DELETES?

Which levels should we put an INSERTED key into?

How can we keep a good spread of keys at each levels?

especially when we don’t know what will be INSERTED and DELETED in the future
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

Flip one coin for each key...
For each key that got a head, put it in the new top level
Repeat with the keys from the new top level
(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level
Repeat with the keys from the new top level
(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

Flip one coin for each key...
For each key that got a head, put it in the new top level
Repeat with the keys from the new top level
(stop when the top level contains only the smallest and largest keys)

(we still always include the smallest and largest keys in every level)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level

Repeat with the keys from the new top level

(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level

Repeat with the keys from the new top level

(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level

Repeat with the keys from the new top level

(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level

Repeat with the keys from the new top level

(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level
Repeat with the keys from the new top level

(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level
Repeat with the keys from the new top level
(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

Flip one coin for each key...

For each key that got a head, put it in the new top level
Repeat with the keys from the new top level
(stop when the top level contains only the smallest and largest keys)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

This doesn’t look quite perfect but actually, it’s very good with high probability (more on this later)
Building Multi-level Linked Lists by flipping coins

Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins

(we still always include the smallest and largest keys in every level)

This doesn’t look quite perfect but actually, it’s very good with high probability (more on this later)

The intuition is that \( n \) coin flips contain about \( \frac{n}{2} \) heads and about \( \frac{n}{2} \) tails
Before we formally introduce Skip Lists, we let’s rewind and try building another Multi-level Linked List... by flipping coins.

This doesn’t look quite perfect but actually, it’s very good with high probability (more on this later). The intuition is that \( n \) coin flips contain about \( \frac{n}{2} \) heads and about \( \frac{n}{2} \) tails and the heads are roughly evenly spread out.
A skip list is a multi-level linked list where the inserts are done by flipping coins 

\[ \text{i.e. this is a skip list...} \]
Skip Lists

A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins

*i.e. this is a skip list.*

To perform **INSERT**(*x*, *k*),

**Step 1:** Use **FIND**(*k*) to insert (*x*, *k*) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert (*x*, *k*) into the next level up
  *(if there is no ‘next level up’, create a new level at the top)*
- If you get a **tails**, **stop**
Skip Lists

A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins

*i.e. this is a skip list...*

To perform **INSERT**(\(x, k\)),

**Step 1:** Use **FIND**(\(k\)) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a heads, insert \((x, k)\) into the next level up
  *(if there is no ‘next level up’, create a new level at the top)*
- If you get a tails, stop
Skip Lists

A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins

*i.e. this is a skip list...*

To perform **INSERT**\((x, k)\),

**Step 1:** Use **FIND**\((k)\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert \((x, k)\) into the next level up
- *if there is no 'next level up', create a new level at the top*

- If you get a **tails**, **stop**
Skip Lists

A skip list is a multi-level linked list where the **INSERT**s are done by flipping coins

*i.e. this is a skip list...*

To perform **INSERT**(*x*, *k*),

**Step 1:** Use **FIND**(*k*) to insert (*x*, *k*) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert (*x*, *k*) into the next level up
  *(if there is no ‘next level up’, create a new level at the top)*
- If you get a **tails**, stop
Skip Lists

A skip list is a multi-level linked list where

the **INSERTS** are done by flipping coins

*i.e. this is a skip list.*

To perform **INSERT** \((x, k)\),

**Step 1:** Use **FIND** \((k)\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

If you get a **heads**, insert \((x, k)\) into the next level up

*(if there is no ‘next level up’, create a new level at the top)*

If you get a **tails**, **stop**
Skip Lists

A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins

\textit{i.e. this is a skip list}...

To perform \texttt{INSERT}(x, k),

**Step 1:** Use \texttt{FIND}(k) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a \texttt{heads}, insert \((x, k)\) into the next level up
  \textit{(if there is no 'next level up', create a new level at the top)}
- If you get a \texttt{tails}, \texttt{stop}
A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins

\[ \text{i.e. this is a skip list} \ldots \]

To perform **INSERT** \((x, k)\),

**Step 1:** Use **FIND** \((k)\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

If you get a **heads**, insert \((x, k)\) into the next level up

\( \text{(if there is no ‘next level up’, create a new level at the top)} \)

If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INSERT**s are done by flipping coins *i.e. this is a skip list*.

To perform **INSERT**\((x, k)\),

**Step 1:** Use **FIND**\(k\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert \((x, k)\) into the next level up
  
  *(if there is no 'next level up', create a new level at the top)*

- If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INSERTs** are done by flipping coins

*i.e. this is a skip list...*

To perform **INSERT**$(x, k)$,

**Step 1:** Use **FIND**$(k)$ to insert $(x, k)$ into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert $(x, k)$ into the next level up
  
  *(if there is no 'next level up', create a new level at the top)*

- If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins. *i.e. this is a skip list.*

To perform \(\text{INSERT}(x, k)\),

**Step 1:** Use \(\text{FIND}(k)\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

1. If you get a **heads**, insert \((x, k)\) into the next level up *(if there is no ‘next level up’, create a new level at the top)*
2. If you get a **tails**, **stop**
A skip list is a multi-level linked list where
the **INSERTS** are done by flipping coins
  *i.e. this is a skip list...

To perform **INSERT**(x, k),

**Step 1:** Use **FIND**(k) to insert (x, k) into the bottom level

**Step 2:** Flip a coin repeatedly:

  If you get a **heads**, insert (x, k) into the next level up
  *(if there is no ‘next level up’, create a new level at the top)*

  If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins 

*i.e. this is a skip list.*

To perform **INSERT** \((x, k)\),

**Step 1:** Use **FIND** \(k\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert \((x, k)\) into the next level up
  
  *if there is no ‘next level up’, create a new level at the top*

- If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins.

_i.e. this is a skip list..._

To perform **INSERT**(*x*, *k*),

**Step 1:** Use **FIND**(*k*) to insert (*x*, *k*) into the bottom level.

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert (*x*, *k*) into the next level up.
  
  _if there is no ‘next level up’, create a new level at the top_

- If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INSERT**s are done by flipping coins

* i.e. this is a skip list...

To perform **INSERT**\((x, k)\),

**Step 1:** Use **FIND**\((k)\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert \((x, k)\) into the next level up
  
  * (if there is no ‘next level up’, create a new level at the top)

- If you get a **tails**, **stop**
A skip list is a multi-level linked list where the \texttt{INSERTS} are done by flipping coins. \textit{i.e. this is a skip list}.

To perform \texttt{INSERT}(x, k),

\textbf{Step 1:} Use \texttt{FIND}(k) to insert (x, k) into the bottom level.

\textbf{Step 2:} Flip a coin repeatedly:

\begin{itemize}
  \item If you get a \texttt{heads}, insert (x, k) into the next level up \textit{(if there is no ‘next level up’, create a new level at the top)}
  \item If you get a \texttt{tails}, \texttt{stop}
\end{itemize}
Skip Lists

A skip list is a multi-level linked list where the **INSERTS** are done by flipping coins

*i.e. this is a skip list...*

To perform **INSERT** \((x, k)\),

**Step 1:** Use **FIND** \((k)\) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert \((x, k)\) into the next level up
  
  *if there is no ‘next level up’, create a new level at the top*

- If you get a **tails**, **stop**
Skip Lists

A skip list is a multi-level linked list where
the **INSERTS** are done by flipping coins

*i.e. this is a skip list...*

To perform **INSERT**\((x, k)\),

**Step 1**: Use **FIND**\((k)\) to insert \((x, k)\) into the bottom level

**Step 2**: Flip a coin repeatedly:

If you get a **heads**, insert \((x, k)\) into the next level up

(*if there is no 'next level up', create a new level at the top*)

If you get a **tails**, **stop**
A skip list is a multi-level linked list where the **INserts** are done by flipping coins *i.e. this is a skip list...*

---

**To perform** \text{INSERT}(x, k),

**Step 1:** Use \text{FIND}(k) to insert \((x, k)\) into the bottom level

**Step 2:** Flip a coin repeatedly:

- If you get a **heads**, insert \((x, k)\) into the next level up
  (*if there is no ‘next level up’, create a new level at the top*)
- If you get a **tails**, **stop**
That about DELETEs?

DELETING is straightforward, just FIND the key and DELETE it from all levels
That about `DELETES`?

`DELETING` is straightforward, just `FIND` the key and `DELETE` it from all levels.

To perform `DELETE(k),`

**Step 1:** Use `FIND(k)` to find `(x, k)`

**Step 2:** Delete `(x, k)` from all levels
DELETING is straightforward, just FIND the key and DELETE it from all levels.

To perform DELETE($k$),

**Step 1:** Use FIND($k$) to find $(x, k)$

**Step 2:** Delete $(x, k)$ from all levels
That about \textbf{DELETES}?

\textbf{DELETING} is straightforward, just \textbf{FIND} the key and \textbf{DELETE} it from all levels.

To perform \textbf{DELETE}(k),

\textbf{Step 1}: Use \textbf{FIND}(k) to find \((x, k)\)

\textbf{Step 2}: Delete \((x, k)\) from all levels.
That about **DELETE**s?

**DELETING** is straightforward, just **FIND** the key and **DELETE** it from all levels.

To perform **DELETE**(\(k\)),

**Step 1:** Use **FIND**\((k)\) to find \((x, k)\)

**Step 2:** Delete \((x, k)\) from all levels
That about **DELETE**?

**DELETE** is straightforward, just **FIND** the key and **DELETE** it from all levels

To perform **DELETE**(\(k\)),

**Step 1:** Use **FIND**(\(k\)) to find \((x, k)\)

**Step 2:** Delete \((x, k)\) from all levels
Skip Lists

That about **DELETE**s?

**DELETING** is straightforward, just **FIND** the key and **DELETE** it from all levels.

To perform **DELETE**\((k)\),

**Step 1:** Use **FIND**\((k)\) to find \((x, k)\)

**Step 2:** Delete \((x, k)\) from all levels.
That about **DELETE**? 

**DELETE**ing is straightforward, just **FIND** the key and **DELETE** it from all levels.

To perform **DELETE**($k$),

**Step 1:** Use **FIND**($k$) to find ($x$, $k$)

**Step 2:** Delete ($x$, $k$) from all levels.
That about **DELETES**?

**DELETING** is straightforward, just **FIND** the key and **DELETE** it from all levels

To perform **DELETE**\((k)\),

**Step 1:** Use **FIND**\((k)\) to find \((x, k)\)

**Step 2:** Delete \((x, k)\) from all levels
That about **DELETE**s?

**Deleting** is straightforward, just **find** the key and **delete** it from all levels

To perform **DELETE**(*k*),

**Step 1:** Use **find**(*k*) to find (*x*, *k*)

**Step 2:** Delete (*x*, *k*) from all levels

**Step 2:** Remove any empty levels

*(ones containing only the smallest and largest keys)*
That about **DELETE**s?

**DELETING** is straightforward, just **FIND** the key and **DELETE** it from all levels.

To perform **DELETE**($k$),

**Step 1:** Use **FIND**($k$) to find ($x$, $k$)

**Step 2:** Delete ($x$, $k$) from all levels

**Step 2:** Remove any empty levels (ones containing only the smallest and largest keys)
Skip Lists

That about **DELETE**s?

**DELETING** is straightforward, just **FIND** the key and **DELETE** it from all levels.

To perform **DELETE**($k$),

**Step 1:** Use **FIND**($k$) to find ($x$, $k$)

**Step 2:** Delete ($x$, $k$) from all levels

**Step 2:** Remove any empty levels

*(ones containing only the smallest and largest keys)*
A skip list is a **randomised** data structure, based on link lists with **shortcuts** which supports **INSERT**($x, k$), **FIND**($k$) and **DELETE**($k$).

We will show that each of these operations takes *expected* $O(\log n)$ time.

That is, they take $O(\log n)$ time ‘on average’

**Important** There is *no randomness in the data*,

the only randomness is in the coin flips.

On the worst case input sequence, the expected time is $O(\log n)$.
How many levels are in a Skip list?

We begin by proving that after $n$ INSERT operations, a skip list is very unlikely to have more than $2\log n$ levels...
How many levels are in a Skip list?

We begin by proving that after $n$ INSERT operations, a skip list is very unlikely to have more than $2 \log n$ levels...

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.
How many levels are in a Skip list?

We begin by proving that after $n$ INSERT operations, a skip list is very unlikely to have more than $2\log n$ levels...

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.

Consider some INSERT($x, k$) operation.
How many levels are in a Skip list?

We begin by proving that after $n$ INSERT operations, a skip list is very unlikely to have more than $2 \log n$ levels.

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.

Consider some INSERT($x, k$) operation.

The probability ($x, k$) is inserted into more than 1 level is $\frac{1}{2}$

(\textit{the first coin flip is H})
How many levels are in a Skip list?

We begin by proving that after \( n \) INSERT operations, a skip list is very unlikely to have more than \( 2 \log n \) levels...

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.

Consider some INSERT\((x, k)\) operation

The probability \((x, k)\) is inserted into more than 1 level is \( \frac{1}{2} \)

\(\text{(the first coin flip is H)}\)

The probability \((x, k)\) is inserted into more than 2 levels is \( \frac{1}{4} \)

\(\text{(we throw HH...)}\)
How many levels are in a Skip list?

We begin by proving that after \( n \) INSERT operations, a skip list is very unlikely to have more than \( 2 \log n \) levels...

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.

Consider some \( \text{INSERT}(x, k) \) operation:

- The probability \((x, k)\) is inserted into more than 1 level is \(\frac{1}{2}\) \((the \ first \ coin \ flip \ is \ H)\).
- The probability \((x, k)\) is inserted into more than 2 levels is \(\frac{1}{4}\) \((we \ throw \ HH\ldots)\).
- The probability \((x, k)\) is inserted into more than 3 levels is \(\frac{1}{8}\) \((we \ throw \ HHH\ldots)\).
How many levels are in a Skip list?

We begin by proving that after \( n \) INSERT operations, a skip list is very unlikely to have more than \( 2 \log n \) levels...

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.

Consider some INSERT\((x, k)\) operation

- The probability \((x, k)\) is inserted into more than 1 level is \(\frac{1}{2}\) (the first coin flip is H)
- The probability \((x, k)\) is inserted into more than 2 levels is \(\frac{1}{4}\) (we throw HH... )
- The probability \((x, k)\) is inserted into more than 3 levels is \(\frac{1}{8}\) (we throw HHH... )
- The probability \((x, k)\) is inserted into more than \(j\) levels is \(\frac{1}{2^j}\)
How many levels are in a Skip list?

We begin by proving that after \( n \) INSERT operations, a skip list is very unlikely to have more than \( 2 \log n \) levels...

An empty skip list contains only one level and the only way this can increase is during an INSERT operation.

Consider some \( \text{INSERT}(x, k) \) operation.

The probability \( (x, k) \) is inserted into more than \( j \) levels is \( \frac{1}{2^j} \).
How many levels are in a Skip list?

We begin by proving that after $n$ `INSERT` operations, a skip list is very unlikely to have more than $2 \log n$ levels...

An empty skip list contains only one level and the only way this can increase is during an `INSERT` operation.

Consider some `INSERT(x, k)` operation.

The probability $(x, k)$ is inserted into more than $2 \log n$ levels is $\frac{1}{2^{2 \log n}}$. 
How many levels are in a Skip list?

We begin by proving that after \( n \) \textsc{insert} operations, a skip list is very unlikely to have more than \( 2 \log n \) levels... 

An empty skip list contains only one level and the only way this can increase is during an \textsc{insert} operation.

Consider some \( \textsc{insert}(x, k) \) operation.

The probability \( (x, k) \) is inserted into more than \( 2 \log n \) levels is \( \frac{1}{2^{2 \log n}} = \frac{1}{n^2} \).
How many levels are in a Skip list?

We begin by proving that after \( n \) \texttt{INSERT} operations, a skip list
is very unlikely to have more than \( 2 \log n \) levels...

An empty skip list contains only one level
and the only way this can increase is during an \texttt{INSERT} operation

Consider some \texttt{INSERT}(\( x, k \)) operation

The probability \((x, k)\) is inserted into more than \(2 \log n\) levels is
\[
\frac{1}{2^{2 \log n}} = \frac{1}{n^2}
\]

The \textbf{union} bound

Let \( E_1, E_2 \ldots E_n \) be events where \( E_j \) occurs with probability \( p_j \)

The probability of at least one \( E_j \) occurring is at most \( \sum_j p_j \)
How many levels are in a Skip list?

We begin by proving that after \( n \) \textsc{insert} operations, a skip list is very unlikely to have more than \( 2 \log n \) levels.

An empty skip list contains only one level and the only way this can increase is during an \textsc{insert} operation.

Consider some \textsc{insert}(\( x, k \)) operation.

The probability \( (x, k) \) is inserted into more than \( 2 \log n \) levels is

\[
\frac{1}{2^{2 \log n}} = \frac{1}{n^2}
\]

The \textbf{union} bound

Let \( E_1, E_2 \ldots E_n \) be events where \( E_j \) occurs with probability \( p_j \)

The probability of at least one \( E_j \) occurring is at most \( \sum_j p_j \)

Let \( E_j \) be the event that the \( j \)-th \textsc{insert} puts its element in more than \( 2 \log n \) levels.
How many levels are in a Skip list?

We begin by proving that after \( n \) INSERT operations, a skip list is very unlikely to have more than \( 2 \log n \) levels.

An empty skip list contains only one level
and the only way this can increase is during an INSERT operation

Consider some \( \text{INSERT}(x, k) \) operation

The probability \((x, k)\) is inserted into more than \( 2 \log n \) levels is

\[
\frac{1}{2^{2 \log n}} = \frac{1}{n^2}
\]

The union bound

Let \( E_1, E_2 \ldots E_n \) be events where \( E_j \) occurs with probability \( p_j \)

The probability of at least one \( E_j \) occurring is at most

\[
\sum_j p_j
\]

Let \( E_j \) be the event that the \( j \)-th INSERT puts its element in more than \( 2 \log n \) levels

The probability of at least one \( E_j \) occurring is at most

\[
\sum_j \frac{1}{n^2}
\]
How many levels are in a Skip list?

We begin by proving that after \( n \) INSERT operations, a skip list is very unlikely to have more than \( 2 \log n \) levels...

An empty skip list contains only one level, and the only way this can increase is during an INSERT operation.

Consider some \( \text{INSERT}(x, k) \) operation.

The probability \((x, k)\) is inserted into more than \(2 \log n\) levels is \(\frac{1}{2^{2 \log n}} = \frac{1}{n^2}\)

**The union bound**

Let \( E_1, E_2 \ldots E_n \) be events where \( E_j \) occurs with probability \( p_j \). The probability of at least one \( E_j \) occurring is at most \( \sum_j p_j \).

Let \( E_j \) be the event that the \( j \)-th \text{INSERT} puts its element in more than \(2 \log n\) levels. The probability of at least one \( E_j \) occurring is at most \( \sum_j \frac{1}{n^2} = \frac{1}{n} \).
How many levels are in a Skip list?

After \( n \) INSERT operations, the probability that a skip list has more than \( 2 \log n \) levels... is at most \( \frac{1}{n} \).
How many levels are in a Skip list?

After \( n \) INSERT operations, the probability that a skip list has more than \( 2 \log n \) levels... is at most \( \frac{1}{n} \)

*It gets better as \( n \) increases!*
So how long does a **FIND** take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability), we can conclude that the number of times we move down is very likely to be $O(\log n)$.

Start at the top-left (the head of the top level)

To perform $\text{FIND}(k)$,

While you haven’t found $k$:

- If the node to the right’s key, $k' \leq k$, Move right
- Else Move down
So how long does a FIND take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$
but how many times do we move right?

Start at the top-left (the head of the top level)
To perform FIND($k$),

While you haven’t found $k$:
  If the node to the right’s key, $k' \leq k$
    Move right
  Else
    Move down
So how long does a `FIND` take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability), we can conclude that the number of times we move down is very likely to be $O(\log n)$

but how many times do we move right?

Start at the top-left (the head of the top level)

To perform `FIND(k)`,

While you haven’t found $k$:

- If the node to the right’s key, $k' \leq k$
  - Move right
- Else
  - Move down

Consider `FIND(35)`
So how long does a \textit{FIND} take? (sketch proof)

As the number of levels is \(O(\log n)\) (with high probability), we can conclude that the number of times we \textbf{move down} is very likely to be \(O(\log n)\)

but how many times do we \textbf{move right}?

Start at the top-left \textit{(the head of the top level)}

To perform \textit{FIND}(\(k\)),

\begin{itemize}
  \item While you haven’t found \(k\):
    \begin{itemize}
      \item If the node to the right’s key, \(k' \leq k\), \textbf{Move right}
      \item Else \textbf{Move down}
    \end{itemize}
\end{itemize}

consider \textit{FIND}(35)
So how long does a **FIND** take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we **move down** is very likely to be $O(\log n)$
but how many times do we **move right**?

Start at the top-left (the head of the top level)

To perform **FIND**($k$),

While you haven’t found $k$:

- If the node to the right’s key, $k' \leq k$
  - Move right

Else
  - Move down
So how long does a FIND take? (sketch proof)

As the number of levels is \(O(\log n)\) (with high probability),
we can conclude that the number of times we move down is very likely to be \(O(\log n)\)
but how many times do we move right?

Start at the top-left (the head of the top level)

To perform FIND\((k)\),

While you haven’t found \(k\):

If the node to the right’s key, \(k' \leq k\)

Move right

Else

Move down
So how long does a FIND take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability), we can conclude that the number of times we move down is very likely to be $O(\log n)$

but how many times do we move right?

Start at the top-left (the head of the top level)

To perform FIND($k$),

While you haven’t found $k$:

If the node to the right’s key, $k' \leq k$
    Move right

Else Move down
So how long does a \texttt{FIND} take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we \texttt{move down} is very likely to be $O(\log n)$

but how many times do we \texttt{move right}?

Start at the top-left (the head of the top level)

To perform \texttt{FIND($k$)},

While you haven’t found $k$:

\begin{itemize}
  \item If the node to the right’s key, $k' \leq k$ 
    \begin{itemize}
      \item Move right
    \end{itemize}
  \item Else 
    \begin{itemize}
      \item Move down
    \end{itemize}
\end{itemize}
So how long does a \textbf{FIND} take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$
but how many times do we move right?

Start at the top-left (the head of the top level)

To perform \textbf{FIND}($k$),

\begin{itemize}
  \item While you haven’t found $k$:
    \begin{itemize}
      \item If the node to the right’s key, $k' \leq k$
        \begin{itemize}
          \item Move right
        \end{itemize}
      \item Else
        \begin{itemize}
          \item Move down
        \end{itemize}
    \end{itemize}
\end{itemize}
So how long does a \texttt{FIND} take? (sketch proof)

As the number of levels is \( O(\log n) \) (with high probability),
we can conclude that the number of times we move down is very likely to be \( O(\log n) \)
but how many times do we move right?

To perform \texttt{FIND}(k),

\begin{itemize}
  \item Start at the top-left (the head of the top level)
  \item While you haven’t found \( k \):
    \begin{itemize}
      \item If the node to the right’s key, \( k' \leq k \)
        \begin{itemize}
          \item Move right
        \end{itemize}
      \item Else
        \begin{itemize}
          \item Move down
        \end{itemize}
    \end{itemize}
\end{itemize}
So how long does a FIND take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$
but how many times do we move right?

Start at the top-left (the head of the top level)

To perform FIND($k$),

While you haven’t found $k$:
  
  If the node to the right’s key, $k' \leq k$
    Move right
  
  Else
    Move down
So how long does a FIND take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability), we can conclude that the number of times we move down is very likely to be $O(\log n)$

but how many times do we move right?

Start at the top-left (the head of the top level)

To perform $\text{FIND}(k)$,

While you haven’t found $k$:

If the node to the right’s key, $k' \leq k$

Move right

Else Move down
So how long does a **FIND** take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability), we can conclude that the number of times we **move down** is very likely to be $O(\log n)$ but how many times do we **move right**?

To perform **FIND**($k$),

Start at the top-left (the head of the top level)

While you haven’t found $k$:

- If the node to the right’s key, $k' \leq k$  
  - Move right

- Else  
  - Move down
So how long does a \texttt{FIND} take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we \textbf{move down} is very likely to be $O(\log n)$
but how many times do we \textbf{move right}?

![Diagram](image.png)

consider $\text{FIND}(35)$
So how long does a **FIND** take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),

we can conclude that the number of times we **move down** is very likely to be $O(\log n)$

but how many times do we **move right**?

How long is this path?
So how long does a \texttt{FIND} take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we \textbf{move down} is very likely to be $O(\log n)$
but how many times do we \textbf{move right}? 

Consider $\texttt{FIND}(35)$

How long is this path?

1. Reverse it
So how long does a **FIND** take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),

we can conclude that the number of times we **move down** is very likely to be $O(\log n)$

but how many times do we **move right**?

Consider **FIND**(35)

How long is this path?

1. Reverse it
So how long does a \texttt{FIND} take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$
but how many times do we move right?

How long is this path?
1. Reverse it
2. Convince yourself this is the same path:

Start at \texttt{k}
While not at the top-left:
  If you can, \texttt{Move up}
  Else \texttt{Move left}
So how long does a \textsc{Find} take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$
but how many times do we move right?

How long is this path?

1. Reverse it
2. Convince yourself this is the same path:
3. Now convince yourself
it takes the same time as this:
\textit{(in expectation)}

Start at $k$
While not at the top-left:
    If (flip a coin)
        Move up
    Else
        Move left
So how long does a Find take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$
but how many times do we move right?

How long is this path?
1. Reverse it
2. Convince yourself this is the same path:
3. Now convince yourself
   it takes the same time as this:
   \textit{(in expectation)}

Start at $k$
While not at the top-left:
\begin{itemize}
\item If (flip a coin)
  \begin{itemize}
  \item Move up
  \end{itemize}
\item Else
  \begin{itemize}
  \item Move left
  \end{itemize}
\end{itemize}
So how long does a FIND take? (sketch proof)

As the number of levels is $O(\log n)$ (with high probability),
we can conclude that the number of times we move down is very likely to be $O(\log n)$

but how many times do we move right? $O(\log n)$ in expectation

Consider $\text{FIND}(35)$

How long is this path?

1. Reverse it
2. Convince yourself this is the same path:
3. Now convince yourself
   it takes the same time as this:
   $\text{(in expectation)}$

Start at $k$
While not at the top-left:
   If (flip a coin)
     Move up
   Else
     Move left
Time complexities

When performing a \texttt{FIND} operation, the number of moves is $O(\log n)$ in expectation.
Time complexities

When performing a **FIND** operation, the number of moves is $O(\log n)$ in expectation as each move takes $O(1)$ time, the expected time complexity is $O(\log n)$.
Time complexities

When performing a FIND operation, the number of moves is $O(\log n)$ in expectation

as each move takes $O(1)$ time, the expected time complexity is $O(\log n)$

The number of levels is also $O(\log n)$ in expectation
Time complexities

When performing a \texttt{FIND} operation, the number of moves is $O(\log n)$ in expectation as each move takes $O(1)$ time, the expected time complexity is $O(\log n)$.

The number of levels is also $O(\log n)$ in expectation.

Both \texttt{INSERT} and \texttt{DELETE} also take expected $O(\log n)$ time. 

\textit{This is because they both call} \texttt{FIND} \textit{and then spend} $O(1)$ \textit{time per level.}
Time complexities

When performing a \texttt{FIND} operation, the number of moves is $O(\log n)$ in expectation as each move takes $O(1)$ time, the expected time complexity is $O(\log n)$.

The number of levels is also $O(\log n)$ in expectation.

Both \texttt{INSERT} and \texttt{DELETE} also take expected $O(\log n)$ time.\footnote{this is because they both call \texttt{FIND} and then spend $O(1)$ time per level}

In fact, all three operations actually take $O(\log n)$ time \footnote{with high probability}

i.e. the probability of an operation taking longer is at most $\frac{1}{n}$.
Time complexities

When performing a \texttt{FIND} operation, the number of moves is $O(\log n)$ in expectation as each move takes $O(1)$ time, the expected time complexity is $O(\log n)$.

The number of levels is also $O(\log n)$ in expectation.

Both \texttt{INSERT} and \texttt{DELETE} also take expected $O(\log n)$ time:

\textit{this is because they both call \texttt{FIND} and then spend $O(1)$ time per level.}

In fact, all three operations actually take $O(\log n)$ time with high probability.

i.e. the probability of an operation taking longer is at most $\frac{1}{n}$

\textit{(this is a stronger claim but proving it is harder)}
A skip list is a **randomised** data structure, based on link lists with **shortcuts**

which supports $\text{INSERT}(x, k)$, $\text{FIND}(k)$ and $\text{DELETE}(k)$

each of these operations takes **expected** $O(\log n)$ time

That is, they take $O(\log n)$ time ‘on average’

**Important** There is **no randomness in the data**, the only randomness is in the coin flips

On the worst case input sequence, the expected time is $O(\log n)$
A **dynamic search structure** supports (at least) the following three operations

- **DELETE**\((k)\) - deletes the (unique) element \(x\) with \(x.key = k\)
- **INSERT**\((x, k)\) - inserts \(x\) with key \(k = x.key\)
- **FIND**\((k)\) - returns the (unique) element \(x\) with \(x.key = k\)

Here are the time complexities of the structures we have seen...

<table>
<thead>
<tr>
<th></th>
<th><strong>INSERT</strong></th>
<th><strong>DELETE</strong></th>
<th><strong>FIND</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Linked List</td>
<td>(O(1))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>(O(n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>2-3-4 Tree</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Red-Black Tree</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>Skip list</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
</tr>
</tbody>
</table>

The time complexities for the Skip list are *expected*, for the others, they are *worst case*