In this lecture we investigate searching methods.

Pseudo-formally we can define the act of searching as follows:
- We have some arbitrary structure $S$ which holds tuples $(K, D)$ of key and data items.
- We take as input a key $K'$ and want to find the tuple $(K, D)$ in $S$ so that $K = K'$.

We briefly look back at hashing and search trees seen before and introduce two new data structures:
- B+ trees. These are the standard data structures for database system and are designed to be efficient when the data is stored on disk.
- Skip lists. A randomised data structures that enables us to perform all the operations of a balanced search tree without the complexity needed for rebalancing.

Question: What choices do we have to hold our data?

- Array $O(1)$
- Hash table $O(1)$ expected complexity if the table is not too full
- Binary tree $O(\log n)$ if the tree is balanced
- Balanced tree $O(\log n)$ in the worst case
- others ...

We usually make comparisons based on number of tuples examined although other factors are obviously important as well:
- Insertion and removal of data items.
- Size of structure and access characteristics.

So given the options, why not just use an array?
- If we only use the indices 1 and 1000 we need to allocate a 1000 element array to get the benefits but only use two elements!

Given this fact, either a hash table or search tree is usually the preferred option:
- It is easy for a tree to deliver content in order, much harder for a hash table.
- A balanced tree or hash table provides a guarantee of performance.

A binary tree is the manifestation of binary search:
- We have $n$ nodes, each node has at most two children.
- To find a node, we start at the root and traverse the tree.

Search Trees – Binary

In the balanced tree, searching for $K$ traverses $H, L, J, K$.
In the worst case straggly tree, this same search takes much longer.
Search Trees – Binary

- If we consider binary search trees of height \( h = \log n \):
  - One can implement all sorts of operations on such trees.
  - For example, search for nodes and insert or delete nodes.
  - The operations mostly have a run-time related to \( h \).
- This is only good if \( h \) is small, otherwise the tree is like a list!
- Balanced trees address this issue by trying to ensure \( h \) stays small even as \( n \) grows large:
  - Red-black and B+ trees are examples of this sort of structure.
  - The worst cases are not nearly as bad as unbalanced trees.

Search Trees – Red-Black

- We take a normal binary tree structure and mark each node with a colour, either red or black.
- The tree is then a red-black tree if some conditions hold:
  - Every node is coloured either red or black.
  - The root node is black.
  - Every leaf node is black.
  - If a node is red, both children are black.
  - For every node, all paths from the node to leaves contain the same number of black nodes.
  - We have to rebalance the tree using rotations and recolouring after every insert or delete.
- Red-black trees have good guaranteed worst case behaviour in terms of cpu operations performed
- However, for very large data sets the important factor is the number of disk accesses
- Not considered to be very simple

Search Trees – B and B+

- The idea of B trees is to cope with the practical problem of very large data sets:
  - We allow multi-way trees rather than only binary trees.
  - The keys in every node are stored in sorted order
  - Each node also contains pointers to its children
  - Each node is sized so that it just fits into a disk block
  - Search is performed in a similar way to 2 – 3 and 2 – 3 – 4 trees by following the correct interval in the key set

\[
\begin{array}{c}
A \\
B \\
C \\
E \\
F \\
G \\
I \\
J \\
K \\
M \\
N \\
O
\end{array}
\]

B trees - properties

- B trees have the following properties
  - All leaves have the same depth
  - Lower and upper bounds on the number of keys a node can contain, given as a function of a fixed integer \( t \):
    - Every node other than the root must have \( \geq (t - 1) \) keys, and \( t \) children.
    - If the tree is non-empty, the root must have at least one key (and 2 children)
    - Every node can contain at most \( 2t - 1 \) keys, so any internal node can have at most \( 2t \) children
    - The above properties imply that the height of a B-tree is no more than \( \log_t((n + 1)/2) \), for \( t \geq 2 \), where \( n \) is the number of keys
    - We increase \( t \) to be as large as possible to reduce the depth of the tree
  - A 2 – 3 – 4 tree is simply a B tree with \( t = 2 \)
  - We still keep the tree balanced by splitting and promoting ...
Search Trees – B and B+

- Re-balancing is based on promoting an item to the parent node. Consider this example where we have an over-full node:

```
  D H N
 /  /  /
A B C E F G I J K L M
```

- The centre item is promoted and the rest split in two:

```
  D H K N
 /  /  /
A B C E F G I J
```

- Note that we might need to repeat this process and that even the root node might split.

Imagine a B tree used to search for words. In this case, the tree captures all the information in each node:

```
  Bat Dog Fig
 /  /  /
Ant Cat Eye Gap
```

A B+ tree improves on this by only storing data in the leaf nodes:

```
  Bat Dog Fig
 /  /  /
Ant Bat Cat Dog Eye Fig Gap
```

Now it is more likely that everything except the data fits in memory.

Skip lists

You may not remember every detail of Red-Black trees and how to maintain their desirable properties. A simpler way of solving the same problem can be found by using a randomised data structure called a Skip list:

- We have seen quicksort before which is a randomised algorithm. This is our first randomised data structure
- Simple randomised dynamic search structure
- Easy to implement
- Maintains a dynamic set of $n$ elements in $O(\log n)$ time per operation in expectation and with high probability
- “Almost always” $O(\log n)$

Start from a sorted linked list

- Search time is $O(n)$ in the worst case

Suppose we had two linked lists of subsets of the elements.
- Each element can appear in one or both lists
- How can we speed up the searches?
Skip lists

The key idea is to consider the different linked lists as express or local lines to get you to your destination
- Express lines connect a few stations
- Local lines connect all stations
- You can change lines at some interchange stations

Skip lists - Searching

- Walk right in top linked list until going right would go too far
- Walk down to bottom linked list
- Walk right in bottom list until element found (or not)

Skip lists - Running time

We analyse the cost of searching in a two level skip list. Call the bottom level $L_2$ and the top level $L_1$
- The worst case search cost is roughly $|L_1| + |L_2|/|L_1|
- Minimised when terms are equal
- $|L_1|^2 = |L_2| = n \Rightarrow |L_1| = \sqrt{n}$
- If $|L_1| = \sqrt{n}$ and $|L_2| = n$ then the total cost is roughly $2\sqrt{n}$
Skip lists - Running time
Let’s try increasing the number of levels assuming everything is perfectly balanced.

- With one level the search cost is \( n \).
- With two levels the search cost is \( 2\sqrt{n} \). (Bottom \( n \), top \( \sqrt{n} \)).
- With three levels the search cost is \( 3\sqrt[3]{n} \). (Bottom \( n \), then \( n^{2/3} \), \( n^{1/3} \)).
- With four levels the search cost is \( 4\sqrt[4]{n} \). (Bottom \( n \), then \( n^{3/4} \), \( n^{2/4} \), \( n^{1/4} \)).
- [...] [omitted]
- With \( \log n \) levels the search cost is \( \log n \cdot \log \sqrt{n} = 2 \log n \).

Skip list with \( \log n \) levels
A skip list with \( \log n \) levels is like a binary search tree. Search for the element 50 follows four links but 99 only requires one.

- We now know what the ideal skip list is like.
- Need to maintain something roughly like this under insertions and deletions.

To insert an element \( x \):

- First search to find where \( x \) fits in the bottom level.
- If it is not there insert it into the linked list. The bottom level always contains all element in the data structure.
- Which other levels should we insert \( x \) into?
  - This is where we use randomness for the first time in the data structure.
  - Toss a (fair) coin. If HEADS then promote \( x \) to the next level up and flip again.
  - Probability of promotion = 1/2
  - On average, 1/2 of the elements will be promoted 0 levels, 1/4 of them 1 level, 1/8 of them 2 levels etc.
  - Does this make the skip list have the structure we need?

Skip list - Example
Construct a skip list from scratch using the elements 81, 50, 17, 10, 25, 99, 3 using a real coin.
See blackboard...
A skip list supports three types of operations

- **SEARCH**(*x*), returns **TRUE** if *x* occurs in the skip list
- **INSERT**(*x*), inserts *x* into the skip list by first searching for *x* and then inserting it at the appropriate levels using random coin tosses if it is not found
- **DELETE**(*x*), searches for *x* and then deletes it from all lists that it occurs in

The running time of each operation is dominated by the time taken to search for *x*

How good are skip lists? (speed/balance)

- Intuitively: Pretty good on average
- In fact: Really, really good, almost always

---

**Lemma**

With high probability, an *n*-element skip list has $O(\log n)$ levels

- Event $E$ occurs with high probability (w.h.p.) if, for any $\alpha \geq 1$, there is an appropriate choice of constants for which $E$ occurs with probability at least $1 - O(1/n^\alpha)$
- Can make error probability $O(1/n^\alpha)$ very small by making $\alpha$ large, e.g., 100
- We will need the “Union Bound”, which says that the probability of the union of $k$ events is less than or equal to the sum of the individual probabilities of the $k$ events.

**Proof.**

Probability of having more than $c \log n$ levels

- $\leq n \cdot P(\text{an individual element is promoted more than } c \log n \text{ times})$
- $(\text{Union Bound})$
- $= n(1/2^{c \log n})$
- $= n(1/n^c)$
- $= 1/n^{c-1}$

□

---

**Theorem**

With high probability, every search in a skip list takes $O(\log n)$ time

The main idea is to analyse backwards, from the bottom level upwards

- We want to count the total number of “up” and “left” moves made by the search until it reaches the top left
- Number of “up” moves $< \text{number of levels}$
- The number of levels $\leq c \log n$ w.h.p by the previous lemma
- The remaining technicality is to prove that the number of coin tosses required to get $c \log n$ HEADS is $\Theta(\log n)$ w.h.p.
- The $\Omega(\log n)$ part is obvious. We omit the proof of the upper bound. See end notes for references that contain it.
Conclusions

- We have re-capped on and investigated some new search structures:
  - B and B+ trees can help real-world performance by considering processor and memory characteristics.
  - By considering real-world factors such as disk access, we can get improved performance.
  - We usually use trees to solve “give me the items in order” or “give me the items between x and y” type problems.
- We also introduced an alternative to balanced search trees called Skip lists and showed:
  - That linked lists can be made efficient by storing “local” and “express” versions.
  - How randomness can greatly simplify a data structure while still giving good performance “with high probability”

Further Reading

- Introduction to Algorithms
  - Chapter 11 – Hash Tables
  - Chapter 12 – Binary Trees
  - Chapter 13 – Red-Black Trees
  - Chapter 18 – B Trees
- Data Structures and Algorithm Analysis in C
  M.A. Weiss.
  - Chapter 4 – Trees
  - Chapter 5 – Hashing
- Skip lists
  - Randomized Algorithms
    Rajeev Motwani, Prabhakar Raghavan
    - A tutorial http://eternallyconfuzzled.com/tuts/skip.html