B-Trees

- B-Trees are balanced search trees designed to work well on disks
- B-Trees are *not* binary trees: each node can have many children
- Each node of a B-Tree contains *several* keys, not just one
- When doing searches, we decide which child link to follow by finding the correct interval of our search key in the key set of the current node.

Outline

- B-Trees
- Skip Lists

Disk Accesses

- Consider any search tree
- The number of disk accesses per search will dominate the run time
- Unless the entire tree is in memory, there will usually be a disk access every time an arbitrary node is examined
- The number of disk accesses for most operations on a B-tree is proportional to the height of the B-tree
- I.e. The info on each node of a B-tree can be stored in main memory
B-Tree Properties

The following is true for every node \( x \):

- \( x \) stores keys, \( \text{key}_1(x), \ldots, \text{key}_l(x) \) in sorted order (nondecreasing).
- \( x \) contains pointers, \( c_1(x), \ldots, c_{i+1}(x) \) to its children.
- Let \( k_i \) be any key stored in the subtree rooted at the \( i \)-th child of \( x \), then \( k_1 \leq \text{key}_1(x) \leq k_2 \leq \text{key}_2(x) \cdots \leq \text{key}_l(x) \leq k_{i+1} \).

Note

- The above properties imply that the height of a B-tree is no more than \( \log_t \frac{n+1}{2} \), for \( t \geq 2 \), where \( n \) is the number of keys.
- If we make \( t \), larger, we can save a larger (constant) fraction over RB-trees in the number of nodes examined.
- A \((2-3-4)\)-tree is just a B-tree with \( t = 2 \).

Example B-Tree

- 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.
We will now show that for any B-Tree with height $h$ and $n$ keys, $h \leq \log_t \frac{n+1}{2}$, where $t \geq 2$.

Consider a B-Tree of height $h > 1$

- **Q1:** What is the minimum number of nodes at depth 1, 2, and 3?
- **Q2:** What is the minimum number of nodes at depth $i$?
- **Q3:** Now give a lowerbound for the total number of keys (e.g. $n \geq ???$)
- **Q4:** Show how to solve for $h$ in this inequality to get an upperbound on $h$

**Splay Trees**

- A Splay Tree is a kind of BST where the standard operations run in $O(\log n)$ amortized time
- This means that over $l$ operations (e.g. Insert, Lookup, Delete, etc), the total cost is $O(l \log n)$
- In other words, the average cost per operation is $O(\log n)$
- However a single operation could still take $O(n)$ time
- In practice, they are very fast

**Skip Lists**

- Technically, not a BST, but they implement all of the same operations
- Very elegant randomized data structure, simple to code but analysis is subtle
- They guarantee that, with high probability, all the major operations take $O(\log n)$ time

**High Level Analysis**

Comparison of various BSTs

- RB-Trees: + guarantee $O(\log n)$ time for each operation, easy to augment, − high constants
- AVL-Trees: + guarantee $O(\log n)$ time for each operation, − high constants
- B-Trees: + works well for trees that won’t fit in memory, − inserts and deletes are more complicated
- Splay Trees: + small constants, − amortized guarantees only
- Skip Lists: + easy to implement, − runtime guarantees are probabilistic only
Which Data Structure to use?

- Splay trees work very well in practice, the “hidden constants” are small
- Unfortunately, they can not guarantee that every operation takes $O(\log n)$
- When this guarantee is required, B-Trees are best when the entire tree will not be stored in memory
- If the entire tree will be stored in memory, RB-Trees, AVL-Trees, and Skip Lists are good

Skip List

- A skip list is basically a collection of doubly-linked lists, $L_1, L_2, \ldots, L_x$, for some integer $x$
- Each list has a special head and tail node, the keys of these nodes are assumed to be $-\text{MAXINT}$ and $+\text{MAXINT}$ respectively
- The keys in each list are in sorted order (non-decreasing)

- Every node is stored in the bottom list
- For each node in the bottom list, we flip a coin over and over until we get tails. For each heads, we make a duplicate of the node.
- The duplicates are stacked up in levels and the nodes on each level are strung together in sorted linked lists
- Each node $v$ stores a search key (key($v$)), a pointer to its next lower copy (down($v$)), and a pointer to the next node in its level (right($v$)).

Skip List

- Technically, not a BST, but they implement all of the same operations
- Very elegant randomized data structure, simple to code but analysis is subtle
- They guarantee that, with high probability, all the major operations take $O(\log n)$ time
To do a search for a key, \( x \), we start at the leftmost node \( L \) in the highest level.

- We then scan through each level as far as we can without passing the target value \( x \) and then proceed down to the next level.
- The search ends either when we find the key \( x \) or fail to find \( x \) on the lowest level.

```c
SkipListFind(x, L){
    v = L;
    while (v != NULL) and (Key(v) != x){
        if (Key(Right(v)) > x)
            v = Down(v);
        else
            v = Right(v);
    }
    return v;
}
```
**Insert**

$p$ is a constant between 0 and 1, typically $p = 1/2$, let $\text{rand}()$ return a random value between 0 and 1

```plaintext
Insert(k){
First call Search(k), let pLeft be the leftmost elem <= k in L_1
Insert k in L_1, to the right of pLeft
i = 2;
while (rand()<= p){
   insert k in the appropriate place in L_i;
}
}
```

**Deletion**

- Deletion is very simple
- First do a search for the key to be deleted
- Then delete that key from all the lists it appears in from the bottom up, making sure to “zip up” the lists after the deletion

**Analysis**

- Intuitively, each level of the skip list has about half the number of nodes of the previous level, so we expect the total number of levels to be about $O(\log n)$
- Similarly, each time we add another level, we cut the search time in half except for a constant overhead
- So after $O(\log n)$ levels, we would expect a search time of $O(\log n)$
- We will now formalize these two intuitive observations

**Height of Skip List**

- For some key, $i$, let $X_i$ be the maximum height of $i$ in the skip list.
- Q: What is the probability that $X_i \geq 2 \log n$?
- A: If $p = 1/2$, we have:

\[
P(X_i \geq 2 \log n) = \left(\frac{1}{2}\right)^{2\log n} = \frac{1}{(2\log n)^2} = \frac{1}{n^2}
\]

- Thus the probability that a particular key $i$ achieves height $2 \log n$ is $\frac{1}{n^2}$
Height of Skip List

- Q: What is the probability that any key achieves height $2\log n$?
- A: We want $P(X_1 \geq 2 \log n \text{ or } X_2 \geq 2 \log n \text{ or } \ldots \text{ or } X_n \geq 2 \log n)$
- By a Union Bound, this probability is no more than $P(X_1 \geq k \log n) + P(X_2 \geq k \log n) + \ldots + P(X_n \geq k \log n)$
- Which equals:
  $$\sum_{i=1}^{n} \frac{1}{n^2} = \frac{n}{n^2} = \frac{1}{n}$$

Expected Space

A trick for computing expectations of discrete positive random variables:
- Let $X$ be a discrete r.v., that takes on values from 1 to $n$
  $$E(X) = \sum_{i=1}^{n} P(X \geq i)$$
- Why???

In-Class Exercise

Q: How much memory do we expect a skip list to use up?
- Let $X_i$ be the number of lists elem $i$ is inserted in
- Q: What is $P(X_i \geq 1)$, $P(X_i \geq 2)$, $P(X_i \geq 3)$?
- Q: What is $P(X_i \geq k)$ for general $k$?
- Q: What is $E(X_i)$?
- Q: Let $X = \sum_{i=1}^{n} X_i$. What is $E(X)$?

Key Point: The height of a skip list is $O(\log n)$ with high probability.
Search Time

- It's easier to analyze the search time if we imagine running the search backwards.
- Imagine that we start at the found node \( v \) in the bottommost list and we trace the path backwards to the top leftmost sentinel, \( L \).
- This will give us the length of the search path from \( L \) to \( v \) which is the time required to do the search.

Backwards Search

```c
SLFback(v){
    while (v != L){
        if (Up(v)!=NIL)
            v = Up(v);
        else
            v = Left(v);
    }
}
```

Backward Search

- For every node \( v \) in the skip list \( \text{Up}(v) \) exists with probability \( 1/2 \). So for purposes of analysis, \( \text{SLFBack} \) is the same as the following algorithm:

```c
FlipWalk(v){
    while (v != L){
        if (COINFLIP == HEADS)
            v = Up(v);
        else
            v = Left(v);
    }
}
```

Analysis

- For this algorithm, the expected number of heads is exactly the same as the expected number of tails.
- Thus the expected run time of the algorithm is twice the expected number of upward jumps.
- Since we already know that the number of upward jumps is \( O(\log n) \) with high probability, we can conclude that the expected search time is \( O(\log n) \).