CIS 313

week of Nov 12

eighth week of the term
B-trees

• very important data structure in computer science
• database indexing, hard disk referencing, MongoDB, ...
• balanced, multi-way search tree
• many slight variations, we will use definition in CLRS text
• idea is that nodes are large and fit into a disk block (minimum amount of data that’s pulled off a hard drive)
• node size parameters (here called t) depend on disk speeds, block sizes, etc.
B-tree specifications

• fixed parameter \( t \), called *minimum degree*
• nodes have between \( t-1 \) and \( 2t-1 \) keys
• so therefore they have between \( t \) and \( 2t \) children
• root is exception: it may have as few as 1 key (2 children)
• all null pointers have the same depth (distance from root)
• a 2-3-4 tree is a B-tree with minimum degree \( t=2 \)
B-tree node format

• each node can have between t and 2t children
• a node might look like
  • \(<P_0, K_1, P_1, K_2, P_2, \ldots, P_{q-2}, K_{q-1}, P_{q-1}>\)
  • for t <= q <= 2t (except for root 2<=q<=2t)
• during a search, we split full nodes
• a node is full when it has 2t-1 keys
node split (shown for t=3)

Split a full node

Node <a b c d e> becomes

Split a full node into parent

Promote c by inserting into parent
B-tree height

- theorem 18.1: if \( n \geq 1 \), then for any B-tree containing \( n \) keys of height \( h \) and minimum degree \( t \geq 2 \), \( h \leq \log_t \frac{n+1}{2} \).

- example: \( t=50 \) and \( n=100,000,000 \)
  - \( h \leq \log_{50} 50,000,000.5 \approx 4.53 \leq 5 \)

- suppose 20 records fit on a page

- without the index to find an item we’d need to search about half the 100,000,000/20=5,000,000, resulting in 2,500,000 disk accesses

- with the index we need at most 5+1=6 disk accesses (5 for the tree nodes and one for the page containing that key’s record)
exercise 18.2-1

Insert into initially empty B-tree of min degree \( t = 2 \) the key values:


After the first 3 values:

\[ \langle F, Q, S \rangle \]

A search to place \( K \) causes a split:

\[ \begin{align*}
\langle Q \rangle \\
\langle F \rangle & \quad \langle S \rangle \\
\end{align*} \]

\[ \begin{align*}
\langle Q \rangle \\
\langle F, K \rangle & \quad \langle S \rangle \\
\end{align*} \]

After which \( K \) is placed:
place C

search for L splits full node:

- place C
  - `<Q>`
  - `<C, F, K>`
  - `<S>`

- search for L splits full node:
  - `<Q>`
  - `<C, F, K>`
  - `<S>`
  - L
  - split
  - `<F, Q>`
  - `<C>`
  - `<K>`
  - `<S>`
  - `<F, Q>`
  - L
  - insert
  - `<C>`
  - `<K, L>`
  - `<S>`
place H, T, V

insert W

search

split

insert
insert M

split

search

split

insert
ETC....
aside: splay trees

- maintain balance kind of
- insertion, deletion, union take $O(lg n)$ amortized time
  - a series of $m$ of these operations on $n$ keys takes total time worst case $O(m*lg n)$
  - possible that one operation takes $O(n)$ time but cannot happen often
- NO balance information needs to be stored at node (balance factor, color)
- used in DNS servers sometimes
splay trees use rotations

- idea is that whenever a node is accessed, it is moved to the root by a series of rotations
  - LOTS OF ROTATIONS!
  - and slightly different ones
- if x is child of root, it is moved upwards with a single rotation
  - called a ZIG rotation
- if x has a (RL or LR) grandparent, it is moved up with a double rotation
  - called a ZIG-ZAG rotation
- if x has a (LL or RR) rotation, then moved up with a special rotation
  - ZIG-ZIG
- idea: zig-zigs and zig-zags tend towards rebalancing
zig-zig
zig-zag
example: find 7
1 2 3 4 5 6 x zig-zig x, and done
merge sort

1. break list of n elements into two halves (time O(1))
2. recursively sort each half (time T(n/2)+T(n/2))
3. merge the two sorted lists (time O(n))

- total time $T(n) = 2T(n/2) + O(n)$
- this can be shown to be $T(n) = O(n \log n)$
- master method (powerful) can solve recurrence relations
The Master Method

If \( T(n) \leq aT\left(\frac{n}{b}\right) + O(n^d) \)
then

\[
T(n) = \begin{cases} 
O(n^d \log n) & \text{if } a = b^d \quad (\text{Case 1}) \\
O(n^d) & \text{if } a < b^d \quad (\text{Case 2}) \\
O(n^{\log_b a}) & \text{if } a > b^d \quad (\text{Case 3}) 
\end{cases}
\]
quicksort: "dual" version of merge sort

1. "unmerge" the array
   1. use Partition method
   2. break into "halves" of small elements and large elements
   3. of course, they may not be same size, just hope so

2. sort each side recursively

3. put the two sides together (no work involved here)

• Partition is $O(n)$
• total is $O(n^2)$ worst case
• but $O(n \log n)$ on average
Partition: the key method

\textbf{Partition}(array }A\text{, int }p\text{, int }r\text{)

1 \quad x ← A[r] \quad \triangleright \text{ Choose } \textbf{pivot}
2 \quad i ← p - 1
3 \quad \textbf{for } j ← p \textbf{ to } r - 1
4 \quad \textbf{do if } (A[j] \leq x)
5 \quad \textbf{then } i ← i + 1
6 \quad \text{exchange } A[i] \leftrightarrow A[j]
7 \quad \text{exchange } A[i + 1] \leftrightarrow A[r]
8 \quad \textbf{return } i + 1
Partition: observation

- in place operation
  - merge requires workspace
  - then need to copy items back into array

- single pass

- can choose other elements for pivot
  - median of first, last, middle elements
  - choose random location in \([p,r]\), use element there as pivot
    - that’s Randomized-Partition
    - highly recommended
what partition does

the sizes of the left and right sides depend on the value of $x$
quicksort easy with partition

QuickSort(p, r)

if p<r then
    q = Partition(p,r)
    QuickSort(p,q-1)
    QuickSort(q+1,r)
quicksort worst case

• suppose partition uses time cn

• let T(n) be the time for quicksort on n elements

• in the worst case, the pivot element is always the largest (or smallest) remaining element

• so T(n) = cn + T(n-1)

• expanding T(n) = cn + T(n-1)
  
  = cn + c(n-1) + T(n-2)
  
  = cn + c(n-1) + c(n-2) + T(n-3) ...
  
  = c(n + (n-1) + (n-2) + ... + 1) + T(0)
  
  = cn(n+1)/2 + T(0)
  
  = O(n^2)
quicksort average case

• in the call to Partition, suppose that each returned value of q is equally likely
• there are n possibilities: q = 1, 2, 3, ..., n
• the recursive calls to QuickSort have sizes q-1 and n-q
• so the average case time is \( T(n) = cn + \frac{1}{n} \sum_{q=1}^{n} (T(q - 1) + T(n - q)) \)
• note that in the sum, each of T(0), T(1), ..., T(n-1) appear twice, so we can rewrite it as
• \( T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} 2T(k) = cn + \frac{2}{n} (T(0) + T(1) + \cdots + T(n - 1)) \)
average case (2)

• rewrite
• \( n \cdot T(n) = cn^2 + 2(T(0) + T(1) + \cdots + T(n-1)) \)
• substitute \( n-1 \) for \( n \)
• \( (n - 1) \cdot T(n - 1) = c(n - 1)^2 + 2(T(0) + T(1) + \cdots + T(n - 2)) \)
• subtract one from the other, and notice that \( T(0), ..., T(n-2) \) cancel
• \( nT(n) - (n - 1)T(n - 1) = cn^2 - c(n - 1)^2 + 2T(n - 1) \)
• simplify, collect terms
• \( nT(n) = c(2n - 1) + (n + 1)T(n - 1) \leq 2cn + (n + 1)T(n - 1) \)
average case (3)

• grind away, use "standard" tricks

• \( T(n) \leq 2c + \frac{(n+1)}{n} T(n - 1) \)

• \( \frac{T(n)}{n+1} \leq \frac{2c}{n+1} + \frac{T(n-1)}{n} \)

• let \( f(n) = \frac{T(n)}{n+1} \) and rewrite the previous line as

• \( f(n) \leq \frac{2c}{n+1} + f(n - 1) \)

• expand (again)

• \( f(n) \leq \frac{2c}{n+1} + f(n - 1) = \frac{2c}{n+1} + \frac{2c}{n} + f(n - 2) \leq \frac{2c}{n+1} + \frac{2c}{n} + \frac{2c}{n-1} + f(n - 3) \leq ... \)
average case (4)

• \( ... f(n) \leq 2c \left( \frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} \right) + f(0) = 2c(H_{n+1}-1) + f(0) = 2c \cdot H_{n+1} + \theta(1) \)

• recall the Harmonic series \( H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \theta(1) \)

• so \( f(n) = 2c \cdot \ln(n+1) + \theta(1) \approx 2.885c \cdot \log_2(n + 1) + \theta(1) \)

• remembering that \( f(n) = \frac{T(n)}{n+1} \) we get

\[ \frac{T(n)}{n+1} = \theta(\lg(n + 1)) \text{ or } T(n) = \theta((n \lg(n + 1)) \]

• DONE!!