**Exercise No. 7**

**B-tree**

**B-Trees**

<table>
<thead>
<tr>
<th>B-Tree Properties</th>
<th>1. Each node x has the following fields:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>o n_x - the number of keys in X</td>
</tr>
<tr>
<td></td>
<td>o leaf(X) - true if x is a leaf and false otherwise</td>
</tr>
<tr>
<td>2. If X is an inner node with n_x keys (K_1,...K_{n_x}) in ascending order, then X has n_x+1 children (C_1,...C_{n_x+1})</td>
<td></td>
</tr>
<tr>
<td>3. If k_i is the i'th key in X, then all keys of C_i are smaller than k_i, and all keys of C_{i+1} are larger than k_i</td>
<td></td>
</tr>
<tr>
<td>4. All the leaves are in the same level (= the height of the tree, h)</td>
<td></td>
</tr>
<tr>
<td>5. t = the minimal rank of a B-Tree (דרגת המינימום). Each node except the root, has at least t-1 keys and at most 2t-1 keys</td>
<td></td>
</tr>
</tbody>
</table>

**Motivation**

B-Trees are used when the data size is extremely big and can't be saved in the main memory but in a secondary memory (hard disk ...). Reading from a disk is relatively slow, but B-Trees ensure that the number of disk accesses will be relatively small.

**Theorem**

If T is a B-Tree with n ≥ 1 keys then \( \Rightarrow \text{height}(T) \leq \log\left(\frac{n+1}{2}\right) \)

**Insert(k)** (Intuition)

Insert in leaf only. Go down from the root to the leaf into which the new key k will be inserted while splitting all full nodes on the path.

**Delete(k)** (Intuition)

Go down from the root to a node containing k while making manipulations on the tree to ensure that the current node X (except the root node) on the search path has at least t keys (the ancestor of X may have only t-1 keys after the manipulations on X). Thus, when the Delete function gets to a node containing k, the node will have at least t keys.

**Delete (Node X, Key k)**

1. If node X has less than t keys (t-1 keys)
   a. If node X has a sibling Y with t keys: lend one key from it (key k goes to the father node of X and Y, replaces key k' such that X and Y are its child pointers and the key k' is added to X).
b. node X has both siblings with only t-1 keys: merge X and one of its siblings Y, while adding the key k from the father node as a median key (X and Y are child pointers of k), into new node W, remove k and pointers to X and Y from the father node, add pointer to the newly created node W to the father node of X and Y.

2. if k is in X and X is an internal node
   a. if the child node Y that precedes k in X has at least t keys:
      - \( k' = \text{Max}(Y) \) //find maximal key k' in subtree rooted in Y
      - Delete(Y, k')
      - replace k with k' in X
   b. symmetrically, if the child node Z that follows k in X has at least t keys
      - \( k' = \text{Min}(Z) \) //find minimal key k' in subtree rooted in Z
      - Delete(Z, k')
      - replace k with k' in X
   c. both Y and Z have t-1 keys
      - merge Y, k and Z into one node W
      - delete k from X and pointers to Y and Z and add instead pointer to W
      - Delete(W, k)

3. else if k is in X and X is a leaf node
   a. delete k from X
   b. return

4. else
   a. find child node \( Y_{i+1} \) of X such that
      - \( \text{key}_i(X) < k < \text{key}_{i+1}(X) \) // a pointer between key_\(i\)(X) and key_\(i+1\)(X) points to \( Y_{i+1} \)
   b. Delete(\( Y_{i+1}, k \))

**Question 1**

Assume that \( t=2 \). Draw the B-tree that will be created after inserting the following elements (in this order) A,B,C,D,G,H,K,M,R,W,Z.

**Solution:**

After a node has 4 elements, it will be split (in here B becomes the root, and the rest of the elements are in its sons).
adding A, B, C, D:

```
      B
     / \  
    A   C D
```

adding G:

```
      B
     / \  
    A   C D G
```

adding H (C D G H is split. D joins its father):

```
     B  D
    / \   
   A   C G H
```

adding K:

```
     B  D
    / \   
   A   C G H K
```

adding M:

```
     B  D  H
    / \   /  
   A  C  G K M
```
adding R (BDH is split. The tree is one level higher):

```
D
  B
  H
  A C G K M R
```

adding W (K M R W is split, M joins H):

```
D
  B
  H M
  A C G K R W
```

Adding Z:

```
D
  B
  H M
  A C G K R W Z
```
**Question 2**

Assume that $t=2$. Draw the tree that will result from deleting the element $G$, and then $M$.

![Tree Diagram]

**Solution:**

Here we suggest solution based on the Delete Algorithm described above in this paper. Namely, the names of the performed cases match the above algorithm. In addition, we mention (in green) the names of the cases according the Deletion algorithm proposed in class (Two algorithms are the same, the names of the cases are different).

**Delete $G$:** root node state is not relevant, node HM has 2 ($=t$) values, node $G$ has $t-1$ keys, execute 1b: the leaf node will have GHK, delete $G$ from the leaf node)
Deleting M:

1. execute 1b: (3b)

   ![Tree Diagram]

   2. execute 2a: (2a)
      
      a. find a maximal key in the left subtree of M (this is K).

      *We will always try first to `take` a key from the left sibling or a left subtree, and only if it's impossible, `take` the key from the right.

      b. delete K from the leaf node HK (then number of keys in the root node is irrelevant and HK has t keys, thus we can delete it right away).
      c. K replaces M.

   ![Tree Diagram]
More examples of deletions:
t=3

Delete 2: (case 1)

Delete 52 (2c)

Delete 15 (1a)
**Question 3**

Given two B-trees $T_1$ and $T_2$, both with parameter $t=2$. Each key in $T_1$ is smaller than each key in $T_2$. Suggest a way to efficiently merge $T_1$ and $T_2$ into a single B-tree $T$.

**Solution:**

First find $h(T_1)$ and $h(T_2)$ in $O(h(T_1)+h(T_2))$ time.  
(*) Note that for $t=2$, the minimum keys limitation always holds.

**Case a: $h(T_1)=h(T_2)$:**

1. $k \leftarrow \frac{\text{maxKey}(T_1) + \text{minKey}(T_2)}{2}$
2. Create a new root with the key $k$ (with null record pointer)
3. $T_1$ will be the left sub-tree of $k$ and $T_2$ will be the right sub-tree.
4. $\text{Btree-Delete}(k)$

**Case b: $h(T_1) \neq h(T_2)$:**

Without loss of generality assume $h(T_1) > h(T_2)$.

1. $k \leftarrow \frac{\text{maxKey}(T_1) + \text{minKey}(T_2)}{2}$
2. Like in $\text{Btree-Insert}$, go down on the rightmost path in $T_1$ ($k$ is larger than all the keys in $T_1$), while splitting all full nodes. Stop at the node $y$, such that $h(y) = h(T_2)+1$. $y$ is not full, i.e., has 2 keys at most.
3. Add $k$ as the largest key in $y$
4. $T_2$ will be the right sub-tree of $k$ in the node $y$.
5. $\text{Btree-Delete}(k)$

Running time: $O(h(T_1)+h(T_2))$

**Note:** we have to extend B-Tree to keep filed "height" in every node.
**Question 4**

B-Tree T has \((10^5+1)\) keys. The maximal number of keys in a node is 17. How many disk accesses are required in the worst case while looking for a certain key?

**Solution:**

The number of disk accesses = the number of levels in T
The worst case is when the number of levels is maximal \(\Rightarrow\)
number of nodes is maximal \(\Rightarrow\) number of keys in each node is minimal

The maximum number of keys in a node is 17 \(\Rightarrow t = 9\) \(\Rightarrow\) the minimum number of keys in each node other than the root is 8. Each node other than the root has at least 9 children.
In the worst case the root has only one key, thus 2 children.

Level 0: 1 node
Level 1: 2 nodes
Level 2: 2*9 nodes
Level 3: 2*9*9 nodes
...

Level i: 2*9\(^{i-1}\) nodes
The number of nodes: \(1 + 2(9^0 + 9^1 + \ldots + 9^{h-1})\)
The root has one key, every other node has 8 keys, therefore the number of keys is:

\[
1 + 8 \times 2(9^0 + 9^1 + \ldots + 9^{h-1}) = 10^2 + 1
\]
\[2(9^h - 1) = 10^5
\]
\[9^h = 50001
\]
\[h \approx 4.924
\]

The height must be an integer, this means that there are some nodes with more than \(t-1\) keys.

\[h = 4
\]

\[h = 4 \Rightarrow \text{number of levels is 5 (we started to count from level 0).}
\]

You can use the theorem studied in class:
\[h \leq \log_9((n+1)/2)\]
and get:
\[h \leq \log_9((10^5+2)/2) \approx 4.924 \Rightarrow h = 4\] (where tree height is the largest depth of a node in \(T\) \(\Rightarrow\) number of levels = 5

**Probability**

**Question 1**
You are taking a 30 question, multiple choice test (five choices per question). The directions say that your grade will equal the number of correct answers minus one-fourth of the number of wrong answers. You are sure you have 20 of the answers correct. On each of the remaining 10 questions, you can definitely eliminate two of the choices. If you choose from the remaining three responses at random, what is your expected grade for the entire test?

**Solution:**
You already have 20 points, for each of the remaining questions you have \(1/3\) chance of getting the correct answer. The expected number of correct answer is \(10 \times 1/3 = 3.333\) and the expected number of wrong answer is 10 minus the number of correct answers (6.667) thus the expected grade is \(20 + 3.333 - 6.667 \times 1/4 = 21.667\).

**Question 2**
You have \(n\) dices with \(n\) faces each (1 to \(n\)). You throw all of them at once. What is the probability of getting exactly 1, 2, 3, \ldots, \(n\), i.e., exactly 1 dice that fell on 1, exactly 1 dice that fell on 2 etc.
**Solution:**
There are n! options to have 1,2,3,……n from total of n^n options. Thus the probability is n!/n^n.

**Question 3**
Suggest an algorithm which randomly permutes a given array in place.

**Solution:**

```c
RandomShuffle(A){
    n=length(A)
    for (i=1 to n)
        swap(A[i], A(Random(i, n)))
    return;
}
```

The algorithm produces permutation uniformly at random. Namely, each of the possible n! permutations has probability 1/n! to be produced by the algorithm. The runtime is O(n) (worst case and not expected!) assuming the Random generates random number in O(1) time. The number of used random coins: O(n).

**Question 4**
Suppose that you want to output 0 with probability 1/2 and 1 with probability 1/2. At your disposal is a procedure Flip_Biased(), that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability 1 - p, where 0 < p< 1, but you do not know what p is. Give an algorithm, Flip_Fair() that uses Flip_Biased() as a subroutine, and returns an unbiased answer, returning 0 with probability 1/2 and 1 with probability 1/2. What is the expected running time of your algorithm as a function of p?

**Solution [of von-Neumann]:**

1. Flip the coin twice.
2. If both tosses are the same (heads-heads or tails-tails), repeat step 1.
3. If the tosses come up heads-tails, count the toss as heads. If the tosses come up tails-heads, count it as tails.

The reason this process produces a fair result is that the probability of getting heads and then tails must be the same as the probability of getting tails and then heads which is p(1-p). By excluding the events of two heads and two tails by repeating the procedure, the coin flipper is left with the only two remaining outcomes having equivalent probability.

Let the probability that the coin lands heads up be p and the probability that the coin lands tails up be q = 1 – p.
On average, how many flips does it take to generate a bit using von Neumann’s method?

Let us develop a general formula for this problem:
If each round takes exactly f flips, and the probability of generating a bit each round is e, then the expected number of total flips t satisfies a simple equation.

If we succeed in the first round, we use exactly f flips.
If we do not, then we have flipped the coin f times, and because it is as though we have to start over from the beginning again, the expected remaining number of flips is still t.

Hence t satisfies \( t = e f + (1-e)(f+t) \). or, after simplifying \( t = \frac{f}{e} \).

Using von Neumann’s strategy, each round requires two flips.
Both a 0 and a 1 are each generated with probability pq, so a round successfully generates a bit with probability 2pq.

Hence the average number of flips required to generate a bit is \( \frac{f}{e} = \frac{2}{2pq} = \frac{1}{pq} = \frac{1}{p(1-p)} \)

For further practice

**Question 5**

Let \( A[1, \ldots, n] \) be an array of n distinct numbers. If \( i < j \) and \( A[i] > A[j] \) then the pair \((i, j)\) is called an inversion of \( A \). Suppose that the elements of \( A \) are a random permutation of \( (1, 2, \ldots, n) \). What is the expected number of inversions in \( A \)?

**Solution:**
Let \( X_{ij} \) be an indicator random variable, for each \( 1 \leq i, j \leq n, \ i < j \).

\[
X_{ij} = \begin{cases} 
1, & \text{if } A[i] > A[j] \\
0, & \text{otherwise} 
\end{cases}
\]

Define \( X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \). \( X \) is a random variable counting the number of inversions in \( A \). Hence, the expected number of inversions of \( A \) is: \( E[X] \)

\[
E[X] = E \left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2}
\]

We have \( \Pr[X_{ij} = 1] = \frac{1}{2} \), because given two distinct random numbers from the \( \{1, \ldots, n\} \), the probability that the first is bigger than the second is \( \frac{1}{2} \) (Number of all pairs: \( n(n-1) \), number of the pairs \( ai, aj \) s.t. \( ai > aj \) is: \( n(n-1)/2 \)).

\[
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} = \left( \frac{n}{2} \right) \frac{1}{2} = \frac{n(n-1)}{4} \text{[Note, we expect half of all the pairs to be inversions, i.e. } \Theta(n^2) \text{ pairs.]} \]
**Question to think about at home:**
Given an array of integers A suggest an algorithm which finds in O(n log n) worst case (not expected) time the number of inversions in A.

**Hint: think about Merge Sort**