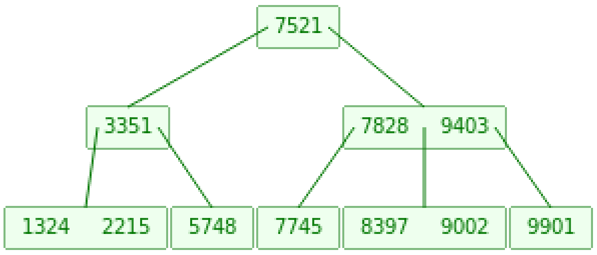
B Tree Using Arrays

Aim:

To represent a B tree as an array in C and implement the functions of :

- Tree creation
- Insertion into the tree
- Searching any value in the tree
- Deletion of a value from the tree

Layout of a sample tree in memory:



This is a sample tree (for t =2) and this is the way it is represented in memory:

Index		L1	V1	L2	V2	L3	V3	V4
0		1	7521	2				
1		3	3351	4				
2		5	7828	6	9403	7		
3		-1	1324	-1	2215	-1		
4		-1	5748	-1				
5		-1	7745	-1				
6		-1	8397	-1	9002	-1		
7		-1	9901	-1				

Creation of tree:

Allocate contiguous memory in the form of an array and initialise al values to 0 and links to -1
Complexity depends only on the size of the array of structures allocated.

Insertion Features:

We always insert the new key into an existing leaf node.
Since we cannot insert a key into a leaf node that is full, we introduce an operation that splits a full node y (having 2t -1 keys) around its median key y_t into two nodes having only t - 1 keys each.
The median key moves up into y's parent to identify the dividing point between the two new trees. But if y's parent is also full, we must split it before we can insert the new key, and thus we could end up splitting full nodes all the way up the tree.

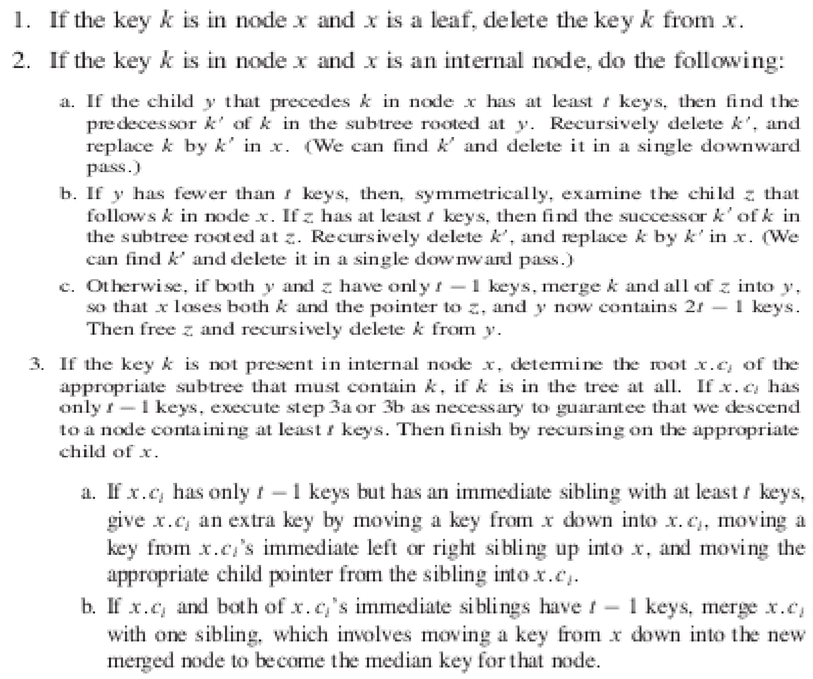
1. Single pass insertion
2. Complexity in case = **O(t * logn)**

Searching Features:

An element can be found either in the leaf or in any internal node. If it is not found in the leaf, that means it doesn't exist.
Reaching a leaf can be checked by checking if the values of the first link in that node is -1
Returns 0 if the element is not found.

1. Single Pass Process
2. Complexity in worst case = **O(t * logn)**

Deletion Features:



1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following:
   a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)
   b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (We can find k' and delete it in a single downward pass.)
   c. Otherwise, if both y and z have only $t - 1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t - 1$ keys. Then free z and recursively delete k from y.
3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ has only $t - 1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.
   a. If $x.c_i$ has only $t - 1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.
   b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t - 1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

1. Single Pass Process
2. Physical deletion (reducing count and shifting elements if deletion happens)
3. Complexity in worst case = **O(t * logn)**

Other Features:

1. The value of degree (t) can be defined in mem_header.h
2. The number of values to be inserted from dataset can be fixed in line 28 of mem_main.c

Execution steps:

```
gcc -c mem_main.c
gcc -c mem_fun.c
gcc mem_main.o mem_fun.o
```

Roshni Venkatesh
01FB15ECS247