Data and File Structures Laboratory

B-Trees, B$^+$-Trees

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

October, 2019

Motivation

Where do search trees (we learned so far) keep the data items?

Motivation

Where do search trees (we learned so far) keep the data items?

We assume that everything in a search tree is kept within the main memory (including the balanced trees like AVL, red-black trees, splay trees, etc.).

Motivation

Where do search trees (we learned so far) keep the data items?

We assume that everything in a search tree is kept within the main memory (including the balanced trees like AVL, red-black trees, splay trees, etc.).

What if the data items contained in a search tree do not fit into the main memory?

Motivation

Just think about searching in the UIDAI database (for AADHAAR details). Let us assume there is only 8 Bytes of data (say the AADHAAR ID and associated details) per citizen and we have to create a search tree.

Motivation

Just think about searching in the UIDAI database (for AADHAAR details). Let us assume there is only 8 Bytes of data (say the AADHAAR ID and associated details) per citizen and we have to create a search tree.

The population of India: 1,370,725,637 (LIVE!!!)

Source: `http://www.worldometers.info/world-population/india-population`

Motivation

Just think about searching in the UIDAI database (for AADHAAR details). Let us assume there is only 8 Bytes of data (say the AADHAAR ID and associated details) per citizen and we have to create a search tree.

The population of India: 1,370,725,637 (LIVE!!!)

Source: `http://www.worldometers.info/world-population/india-population`

The search tree will require more than 20 GB memory (including pointers)!!!

Cycles to access different types of storage

Storage type	Access type	Number of cycles
CPU registers	Random	1
L1 cache	Random	2
L2 cache	Random	30
Main memory	Random	2.5×10^2
Hard disk	Random	3×10^7
	Streamline	5×10^3

Access times (to read or write) on disks are much costlier than the main memory!!!

Search trees on disks

A majority of the tree operations (search, insert, delete, etc.) will require $O(\log_2 n)$ disk accesses where n is the number of data items in the search tree.

The main challenge is to reduce the number of disk accesses for processing per data item.

An m-ary search tree allows m-way branching. As branching increases, the depth decreases. A complete binary tree has a height of $\lceil \log_2 n \rceil$ but a complete m-ary tree has a height of $\lceil log_m n \rceil$.

Characteristics of B-Trees

B-Tree is a low-depth self-balancing tree. The height of a B-Tree is kept low by putting maximum possible keys in a B-Tree node.

The B-Trees have a higher branching factor (also termed as the order) to reduce the depth.

Characteristics of B-Trees

B-Tree is a low-depth self-balancing tree. The height of a B-Tree is kept low by putting maximum possible keys in a B-Tree node.

The B-Trees have a higher branching factor (also termed as the order) to reduce the depth.

Note: Generally, the node size of a B-Tree is kept equal to the disk block size.

B-Trees

Definition (B-Tree)

A B-Tree of order m is an m-ary tree with the following properties:

1. The data items are stored at leaves.
2. The non-leaf nodes store up to $m-1$ keys to guide the searching; The key i represents the smallest key in subtree $i+1$.
3. The root is either a leaf or has between 2 and m children.
4. All non-leaf nodes (except the root) have between $\lceil m/2 \rceil$ and m children.
5. All leaves are at the same depth and have between $\lceil k/2 \rceil$ and k data items, for some k.

B-Trees

Definition (B-Tree)

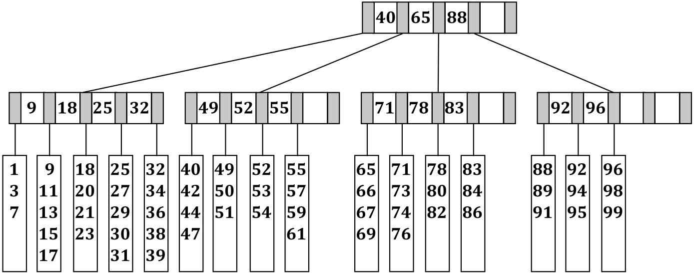
A B-Tree of order m is an m-ary tree with the following properties:

1. The data items are stored at leaves.
2. The non-leaf nodes store up to $m-1$ keys to guide the searching; The key i represents the smallest key in subtree $i+1$.
3. The root is either a leaf or has between 2 and m children.
4. All non-leaf nodes (except the root) have between $\lceil m/2 \rceil$ and m children.
5. All leaves are at the same depth and have between $\lceil k/2 \rceil$ and k data items, for some k.

Note: The properties 3 and 5 are relaxed for the first k insertions.

B-Trees

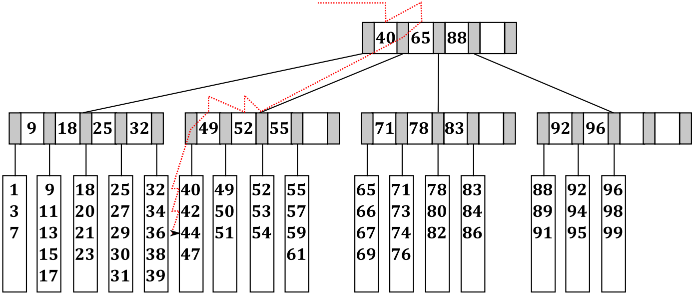
A B-Tree of order 5 and depth 3 that contains 59 data items.



Note: Here, $m = k = 5$.

Searching into B-Trees

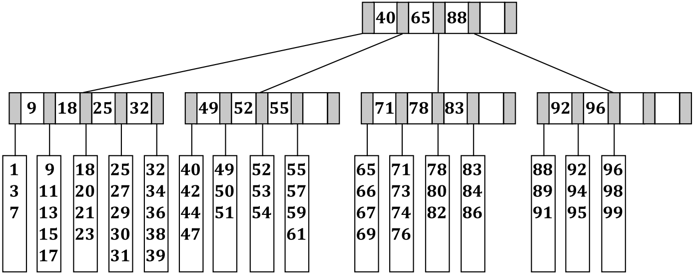
Searching 44 in the following B-Tree:



Note: The lookup (traversal shown in red) is over the disk.

Insertion into B-Trees

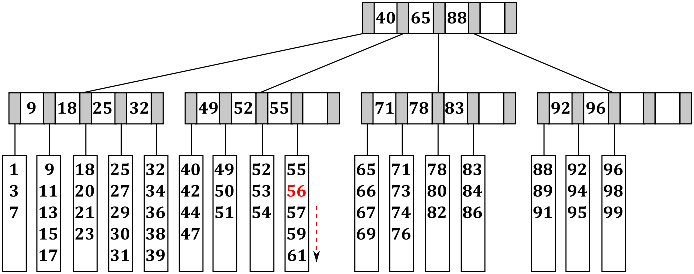
Inserting 56 into the following B-Tree:



<u>**Note**</u>**:** Insertion requires shifting of a few data items.

Insertion into B-Trees

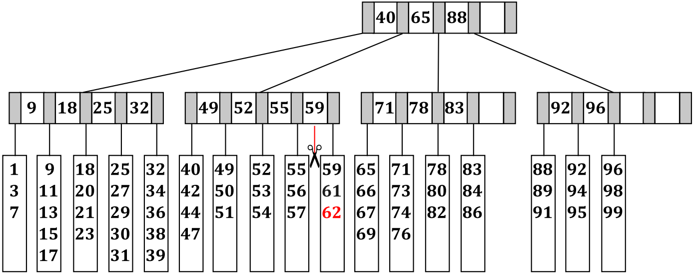
Inserting 62 into the following B-Tree:



Note: Insertion requires breaking a leaf node into a pair of nodes.

Insertion into B-Trees

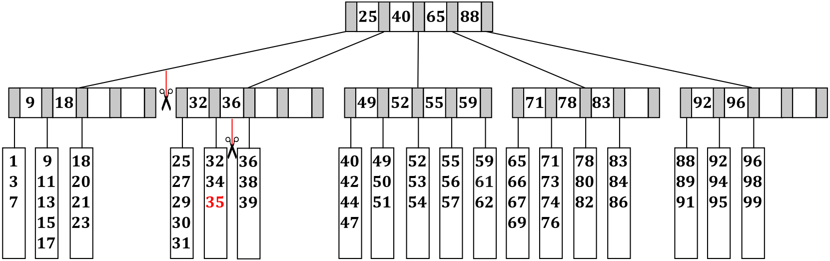
Inserting 35 into the following B-Tree:



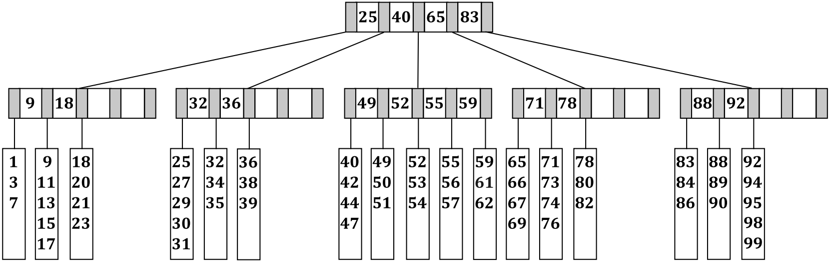
Note: Insertion requires breaking a leaf node into a pair of nodes and the inclusion of a new non-leaf node.

Deletion from B-Trees

Deleting 96 from the following B-Tree:



Deletion from B-Trees



Note: Deletion requires merging of a leaf node with another node.

Conventional implementation of B-Trees

- Individual dynamic memory allocation per node.
- The allocated memories (to the nodes) may be scattered all over the disk making streamline access impossible.

```
typedef int DATA; // OR: typedef void * DATA;

typedef struct treeNode{
    int Count;
    DATA d[ORDER]; // ORDER is a global variable
    struct treeNode *link[ORDER];
    struct treeNode *parent; // This is optional
}BTREENODE;
```

Note: The token `DATA` symbolizes the data type accommodated.

Helper functions

```
extern int compare(BTREENODE *n, DATA d);

extern void print_Btree(BTREENODE *root, int indent);

extern void *insert(BTREENODE *root, DATA d);

extern void *delete(BTREENODE *root, DATA d);

extern BTREENODE *search(BTREENODE *root, DATA d);
```

Searching into B-Trees

```
BTREENODE *search(BTREENODE *root, DATA d) {
    if(root == NULL)
        return NULL;
    i = 0;
    while(i+1 < ORDER)
        if(compare(root, d) < 0)
            return search(root->link[i++], d);
    if(compare(root, d) > 0)
        return search(root->link[i], d);
    else if(0 == compare(root, d))
        return root;
}
```

Note: Every node has more number of links than the number of data items.

Alternative implementation of B-Trees

- Initial dynamic memory allocation and successive reallocations.
- The allocated memories (to the nodes) are located in a contiguous location on the disk.

```
typedef struct treeNode{
    DATA d[ORDER]; // ORDER is a global variable
    int link[ORDER];
    int parent; // This is optional
}BTREE_NLEAFNODE;
typedef struct treeNode{
    DATA d[K];
    int parent; // This is optional
}BTREE_LEAFNODE;
```

Note: `ORDER` and `K` are not necessarily the same.

Alternative implementation – An example

Initially: root = `NULL` (Say `ORDER` = 3)

	Count	d[0]	d[1]	d[2]	link[0]	link[1]	link[2]
free $\rightarrow$ 0	–	–	–	–	NULL	NULL	NULL
1	–	–	–	–	NULL	NULL	NULL
2	–	–	–	–	NULL	NULL	NULL
3	–	–	–	–	NULL	NULL	NULL
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
$n-1$	–	–	–	–	NULL	NULL	NULL

Note: The value `NULL` is treated as '-1' during implementation.

Characteristics of B^+-Trees

Unlike the B-Trees, a B^+-tree does not have data items in the internal (non-leaf) nodes.

Interestingly, more number of keys can be fit on a page of memory in B^+-Trees (because no data is associated with internal nodes), resulting into fewer cache misses in order to access data that is on a leaf node.

B^+-Trees

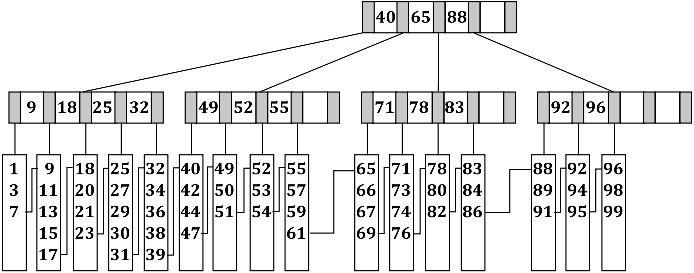
Definition (B^+-Tree)

A B^+-Tree of order m is an m-ary tree with the following properties:

- The data items are stored at leaves.
- The non-leaf nodes store up to $m-1$ keys to guide the searching; The key i represents the smallest key in subtree $i+1$.
- The root is either a leaf or has between 2 and m children.
- All leaves are at the same depth and have up to k data items, for some k.

B$^+$-Trees

A B$^+$-Tree of order 5 and depth 3 that contains 59 data items.



B-Trees vs B^+-Trees

B-Trees	**B^+-Trees**
1. Data are stored on leaf nodes as well as in internal nodes	1. Data are stored on the leaf nodes only
2. Redundant search keys are not allowed	2. Redundant search keys are allowed
3. Leaf nodes are not linked together	3. Leaf nodes are linked together
4. Only direct access is possible	4. Both sequential and direct access are possible
5. Searching is slower	5. Searching is faster
6. Deletion of internal nodes are complicated	6. Deletion of internal nodes are easy

Problems – Day 23

1. Construct a B-Tree of given order with alternative implementation.
2. Write a program that can take a tree and a branching factor as user input and verify whether it is a B-Tree or not.
3. Write a program that implements a function `InOut(struct *, int, int)` that receives a tree and a pair of integers to be inserted into and deleted from the tree, respectively. The function returns 1, if the said insertion and deletion introduce a new leaf node, or 0, otherwise.