What Is a B-tree?

B-trees are a type of self-balancing tree structure designed for storing huge amounts of data for fast query and retrieval. They can be often confused with their close relation – the Binary Search Tree. Although they're both a type of *m*-way search tree, the Binary Search Tree is considered to be a special type of B-tree.

B-trees can have multiple key/value pairs in a node, sorted ascendingly by key order. We call this key/value pair a *payload*. Occasionally we may hear the value part of the payload being referred to as "satellite data" or just "data".

In the context of databases, a key may be the primary key or indexed column of a row, and the value could be the actual row record itself or a reference to it.

In addition to the payload, nodes also keep references to their children. Since each node typically takes up a disk page, these child references usually refer to the page ID in which the child node lives in secondary storage. As any search tree, B-trees are organized so that the keys of any sub-tree must be larger than the keys of the sub-tree to its left.

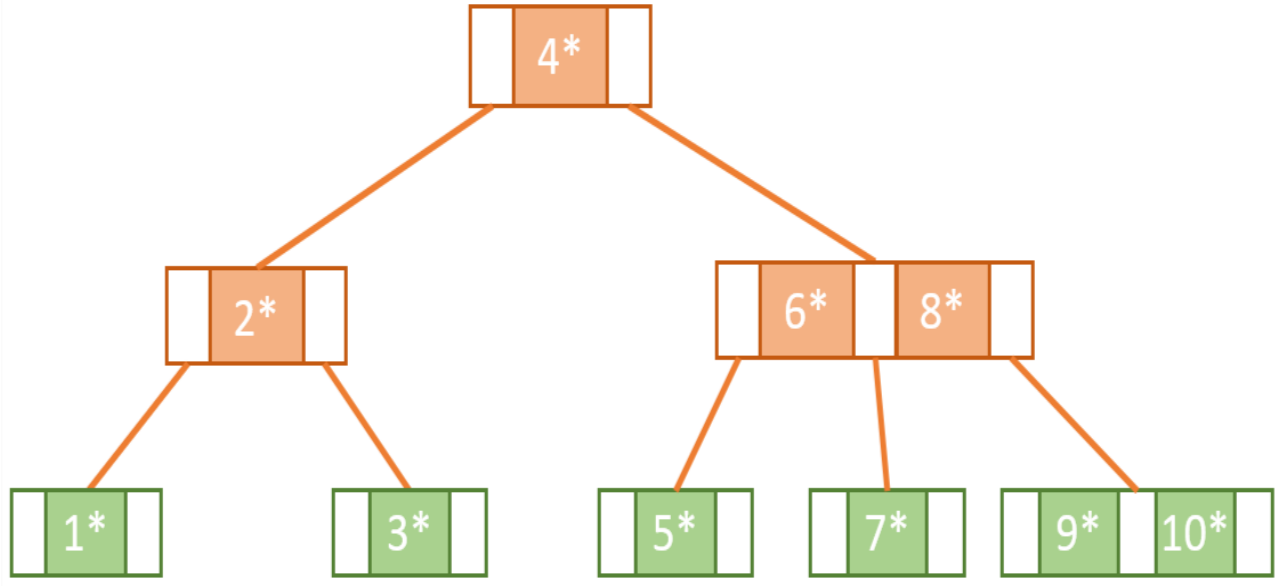
2.1. Properties of a B-tree

For a tree to be classified as a B-tree, it must fulfill the following conditions:

- the nodes in a B-tree of order m can have a maximum of m children

- each internal node (non-leaf and non-root) can have at least (m/2) children (rounded up)
- the root should have at least two children – unless it's a leaf
- a non-leaf node with k children should have k-1 keys
- all leaves must appear on the same level

The image below shows us an example of a B-tree. The maximum number of children in this tree is 3, so we can conclude that this is a tree of order 3. All the leaves are on the same level, and the root and internal nodes have a minimum of two children fulfilling all the conditions of a B-tree:



2.2. Building a B-tree

Now, let's build our tree from scratch. To do this, we'll learn how to insert items into the tree while keeping it balanced.

Since we're starting with an empty tree, the first item we insert will become the root node of our tree. At this point, the root node has the key/value pair. The key is 1, but the value is depicted as a star to

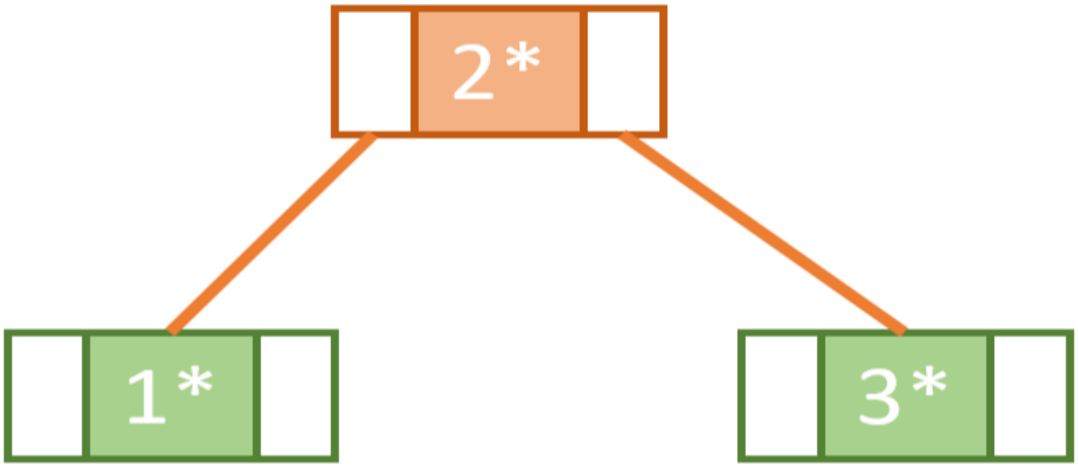
make it easier to represent, and to indicate it is a reference to a record.

The root node also has pointers to its left and right children shown as small rectangles to the left and right of the key. Since the node has no children, those pointers will be empty for now:

We know that this tree has order of 3, so it can have only up to 2 keys in it. So we can add the payload with key 2 to the root node in ascending order:

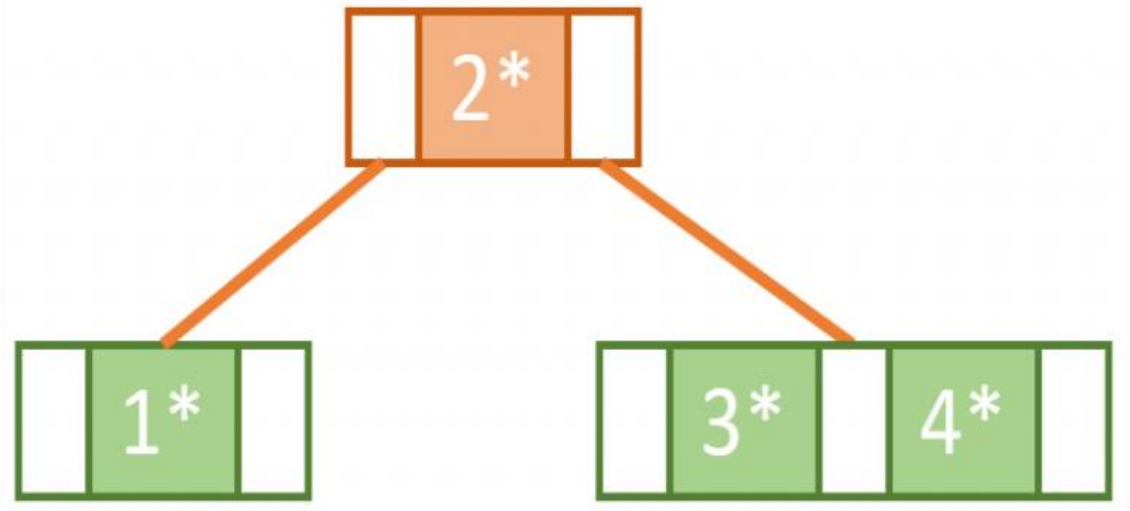
Next, if we wanted to insert 3, for us to keep the tree balanced and fulfilling the conditions of a B-tree we need to perform what is called a split operation.

We can determine how to split the node by picking the middle key. When we choose the middle key we need to consider the keys already present in the node and also the key which is to be inserted. In this case, 2 is the middle key used to split the node. This means the values to the left of 2 will go to a left sub-tree and the values to the right of 2 will go to a right sub-tree and 2 itself will get promoted as part of a new root node:



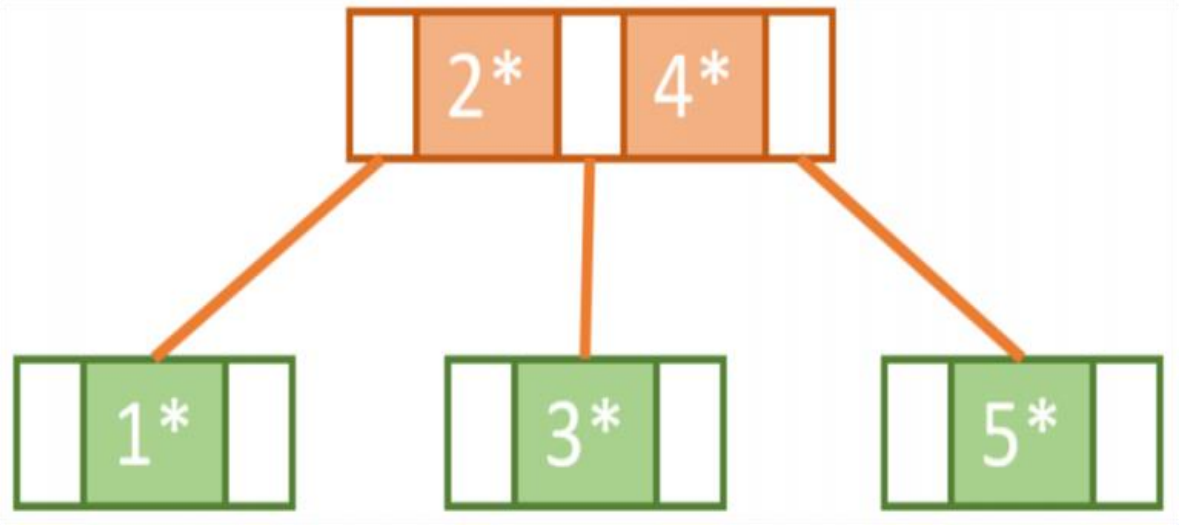
By performing this splitting operation each time we are about to exceed the maximum number of keys in a node, we keep the tree self-balancing.

Now, let's insert 4. To determine where this needs to be placed we must remember that B-trees are organized such that sub-trees on the right have greater keys than sub-trees on the left. Consequently, Key 4 belongs in the right sub-tree. And since the right sub-tree still has the capacity, we can simply add 4 to it alongside 3 in ascending order:

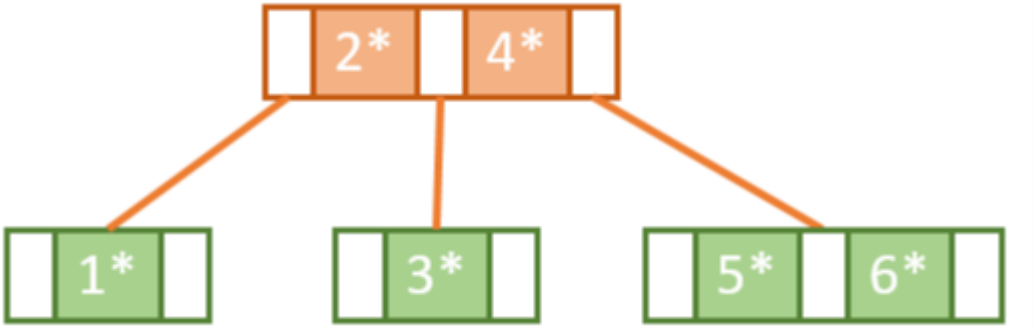


Our right sub-tree is now at full capacity, so to insert 5 we need to use the same splitting logic explained above. We split the node into

two so that Key 3 goes to a left sub-tree and 5 goes to a right sub-tree leaving 4 to be promoted to the root node alongside 2.



This rebalancing gives us space in the rightmost sub-tree to insert 6:

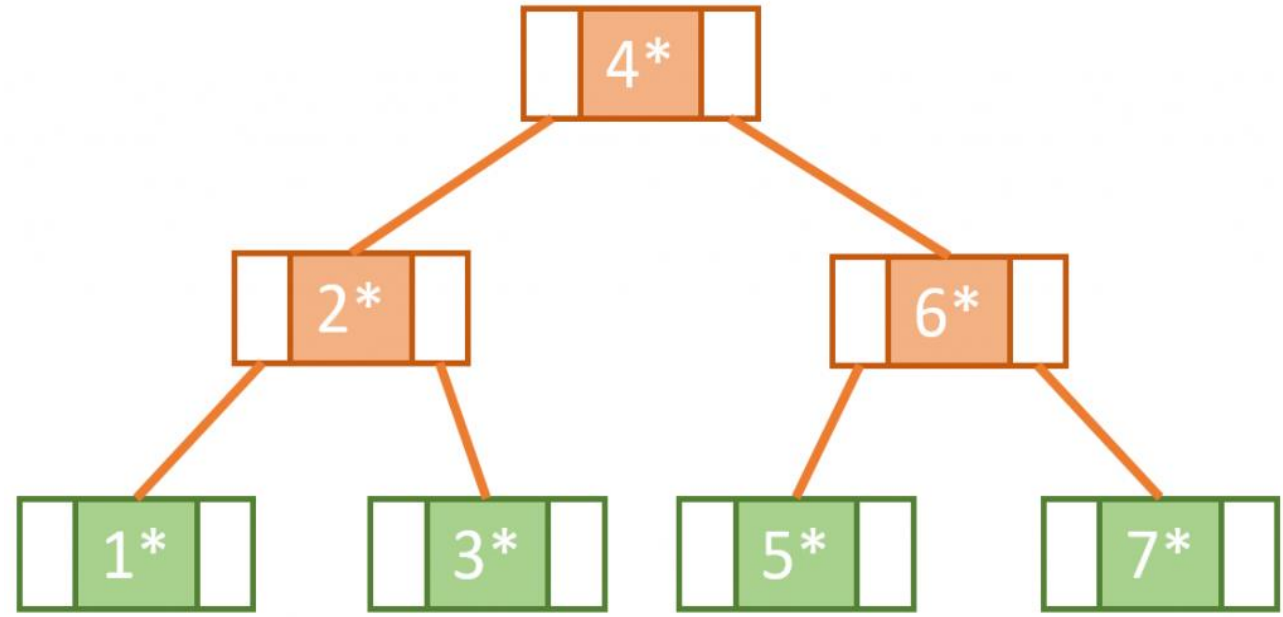


Next, we try to insert 7. However, since the rightmost tree is now at full capacity we know that we need to do another split operation and promote one of the keys.

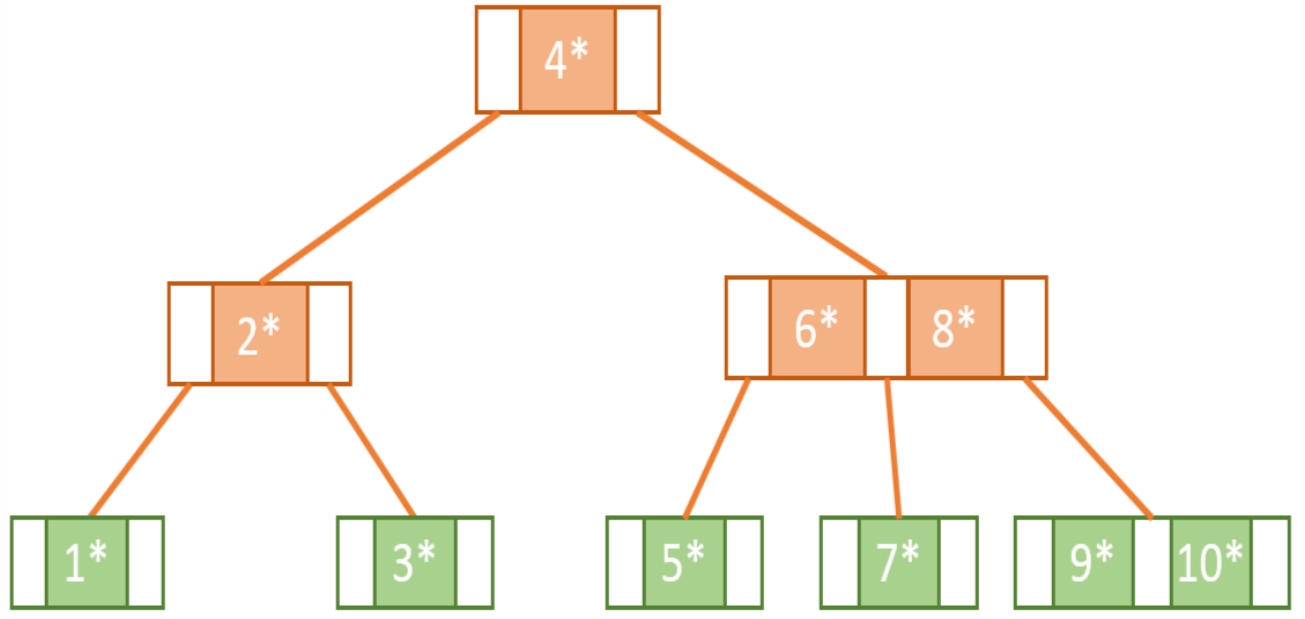
But wait! The root node is also at full capacity, which means that it also needs to be split.

So, we end up doing this in two steps. First, we need to split the right nodes 5 and 6 so that 7 will be on the right, 5 will be on the left, and 6 will be promoted.

Then, to promote 6, we need to split the root node such that 4 will become a part of new root and 6 and 2 become the parents of the right and left subtrees:



Continuing in this way, we fill the tree by adding Keys 8,9 and 10 until we get the final tree:



2.3. Searching a B-tree

B-trees are considered to be so advantageous because they provide logarithmic time complexity when it comes to insert, delete, and search operations.

Let’s take a look at how we would perform a search operation on our B-tree using the pseudocode below:

Algorithm 1: B-tree Search

```
Function Search(node, searchKey):
    i = 0;
    while i < node.Payloads.Count and
      searchKey > node.Payloads[i].Key do
        i++;
    end
    if searchKey = node.Payloads[i].Key then
        return node.Payloads[i].Value;
    end
    if node.IsLeaf = true then
        return null;
    end
    return Search(node.Children[i], searchKey);
end
```

Our pseudocode assumes the following:

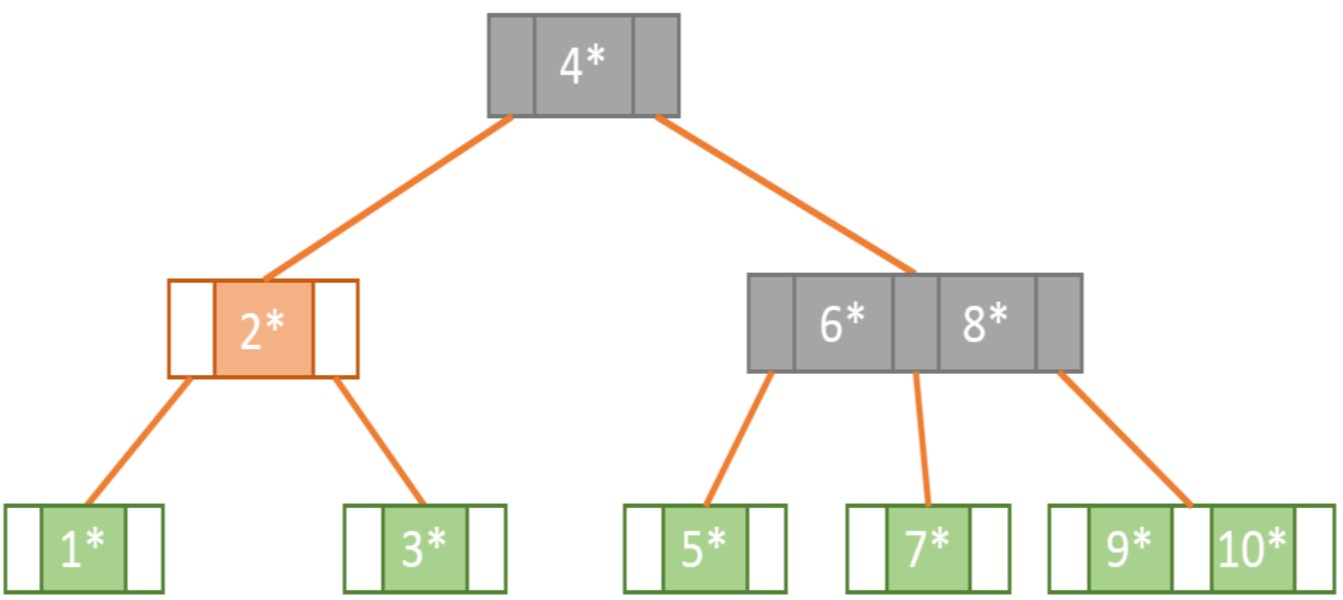
- A node has an array of keys/value pairs *node.Payloads* and an array of references to its children nodes *node.Children*
- The node object has a *node.IsLeaf* property which determines if its a leaf node or not
- *node.Payloads.Count* refers to the number of payloads in a node

Let's see how we can apply this pseudocode if we assume that the keys of our tree refer to IDs of a table, and we'd like to search for the record whose ID is 6. That means we'd be calling the *Search* method defined in the pseudocode with the parameters *node* = root node of the B-tree and *searchKey* = 6.

Using a while loop, we iterate through the root node keys increasing our counter value *i* until we find a key that is greater than or equal to 6 or until we have finished the keys of that node. The loop ends with *i* being at the value 1.

Now using *i = 1* as an index, we select *node.Children[1]* (the child node at index 1). Iterating through its keys, we find *node.Payload[0].Key* to be an exact match to our *searchKey*. We end the search by returning *node.Payload[0]*.Value, which is the satellite data associated with key 6.

The image below shows our search path shaded in grey:

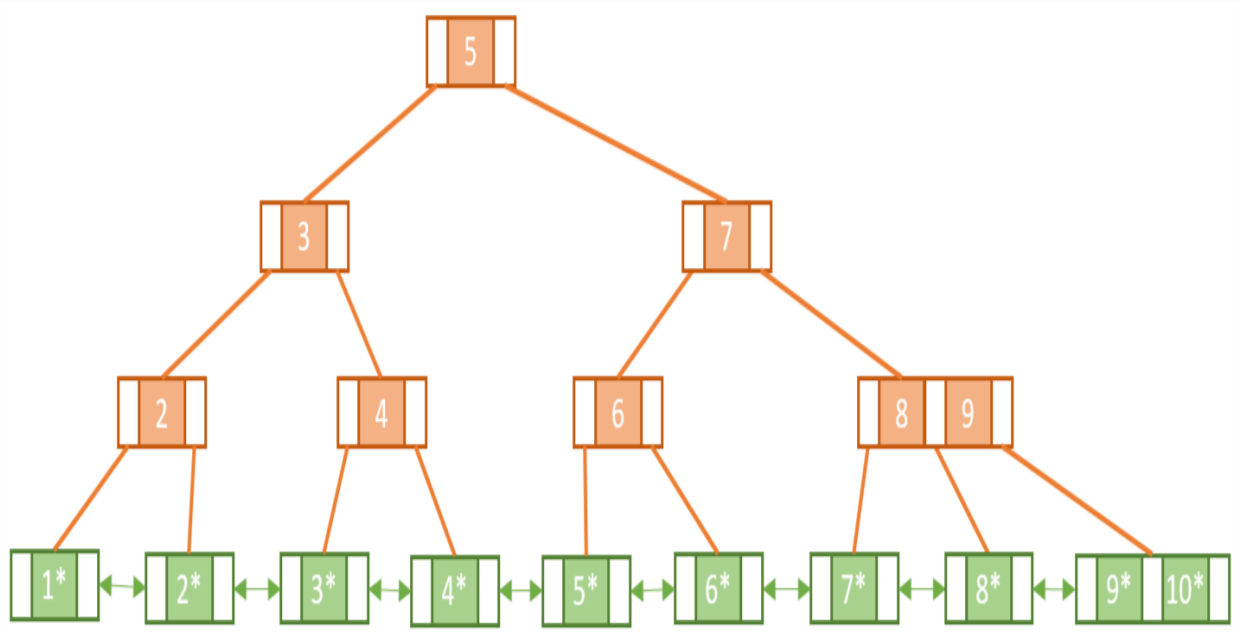


3. The B+tree

The most well-known version of the B-tree is the B+tree. What distinguishes the B+tree from the B-tree are two main aspects:

- all leaf nodes are linked together in a doubly-linked list
- satellite data is stored on the leaf nodes only. Internal nodes only hold keys and act as routers to the correct leaf node

Let’s see how our B-tree would look like if we were to represent its content as a B+tree:



Notice that some keys seem to be duplicated, like 2, 4, and 5. This is because, unlike leaf nodes, the internal nodes of a B+tree cannot hold satellite data and so we must make sure in some way that the leaf nodes include all the keys/value pairs.

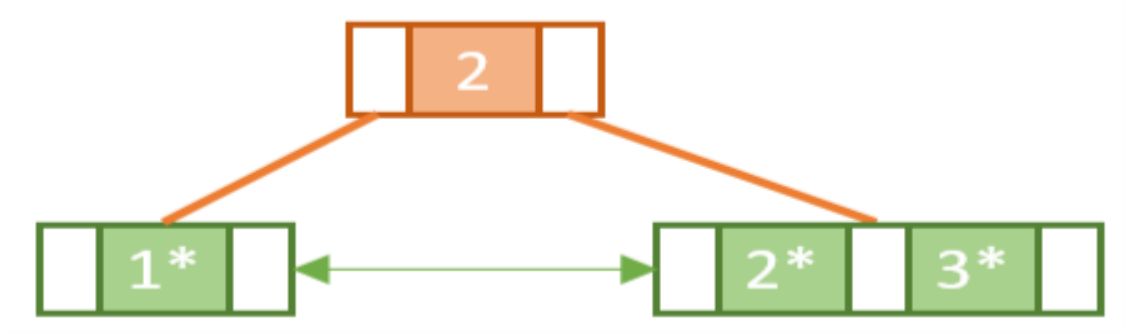
Let's build up our tree from scratch and see how this happens.

3.1. Building a B+tree

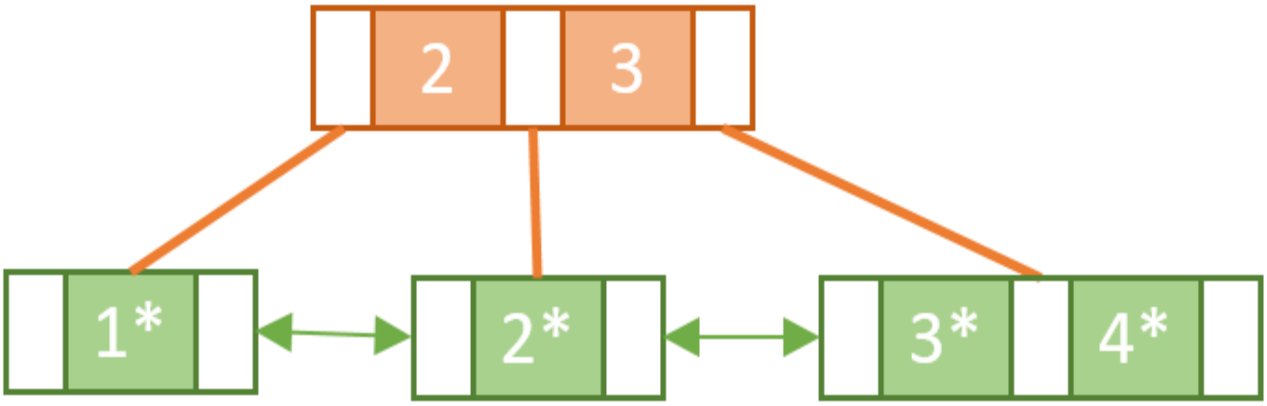
To start with, we'll insert Keys 1 and 2 into the root node in ascending order:

When we come to insert Key 3, we find that in doing so we will exceed the capacity of the root node.

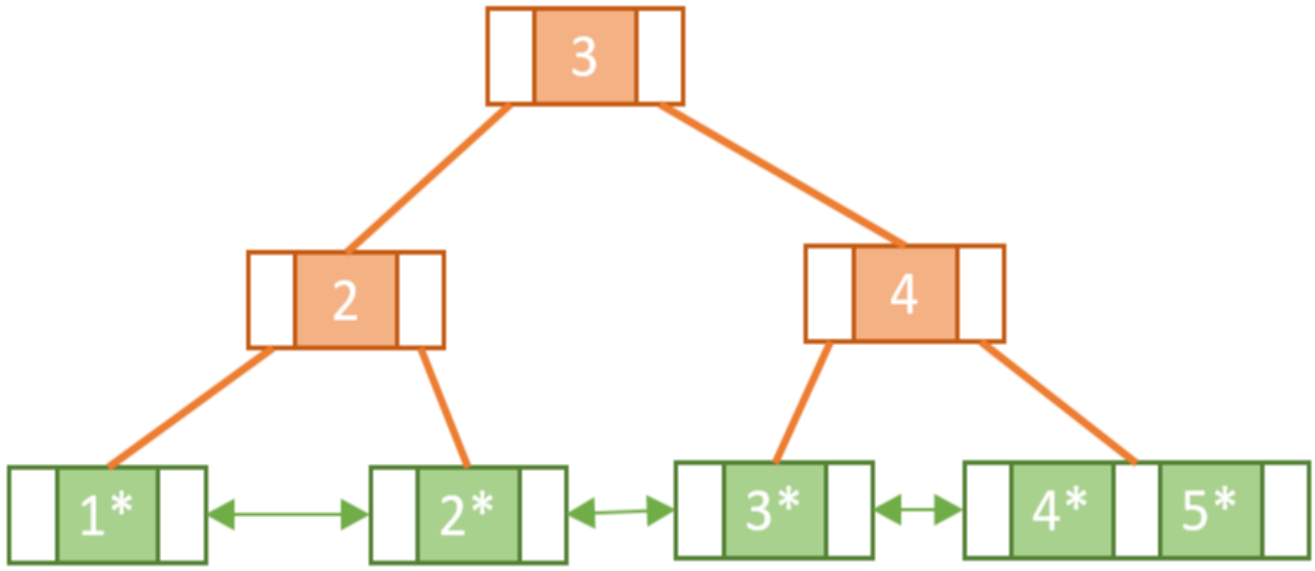
Similar to a normal B-tree this means we need to perform a split operation. However, unlike with the B-tree, we must copy-up the first key in the new rightmost leaf node. As mentioned, this is so we can make sure we have a key/value pair for Key 2 in the leaf nodes:



Next, we add Key 4 to the rightmost leaf node. Since it's full, we need to perform another a split operation and copy-up Key 3 to the root node:

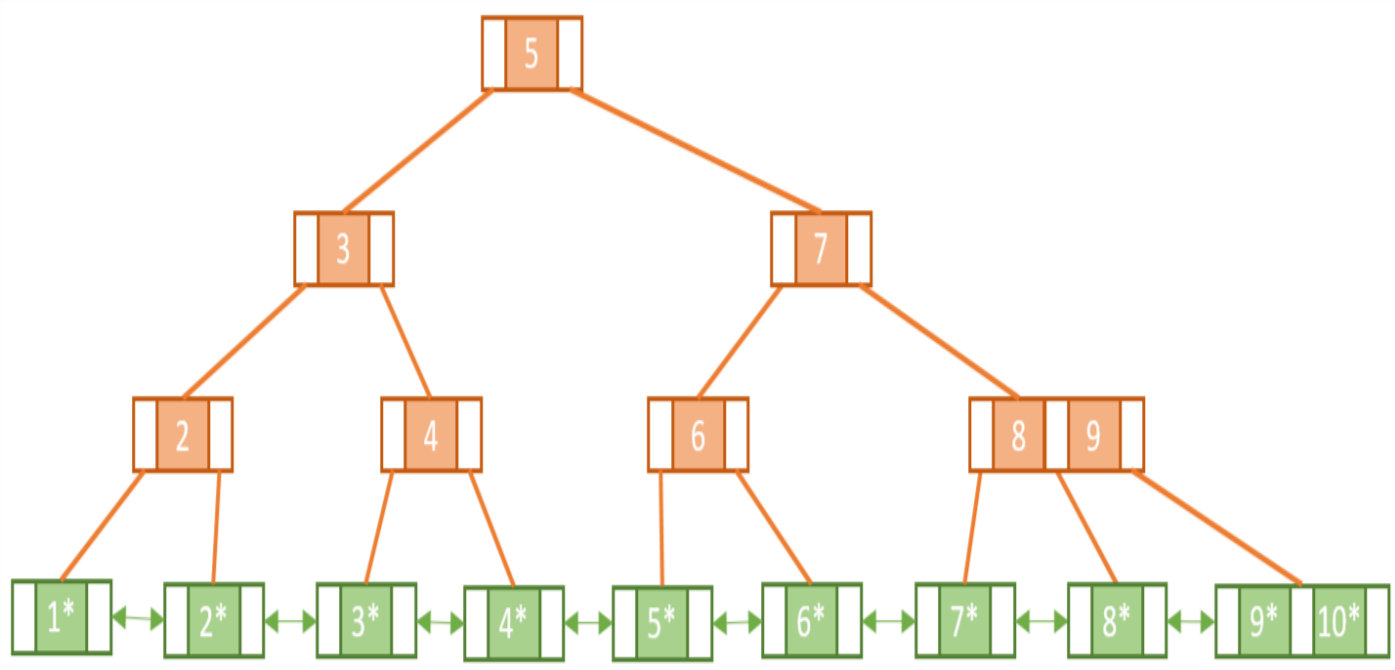


Now, let's add 5 to the rightmost leaf node. Once again to keep the order, we'll split the leaf node and copy-up 4. Since that will overflow the root node, we'll have to perform another split operation splitting the root node into two nodes and promoting 3 into a new root node:



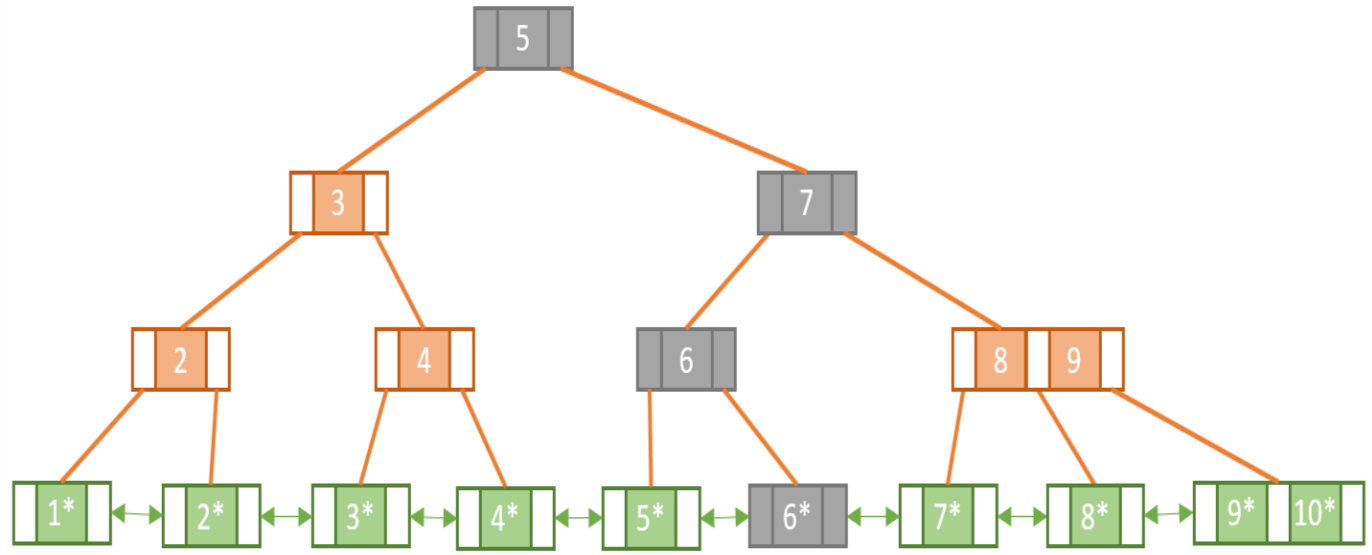
Notice the difference between splitting a leaf node and splitting an internal node. When we split the internal node in the second split operation we didn't copy-up Key 3.

In the same way, we keep adding the keys from 6 to 10, each time splitting and copying-up when necessary until we have reached our final tree:



3.2. Searching a B+tree

Searching for a specific key within a B+tree is very similar to searching for a key in a normal B-tree. Let's see what it would be like to search for Key 6 again but on the B+tree:



The shaded nodes show us the path we have taken in order to find our match. **Deduction tells us that searching within a B+tree means that we must go all the way down to a leaf node to get the satellite data. As opposed to B-trees where we could find the data at any level.**

In addition to exact key match queries, B+trees support range queries. This is enabled by the fact that the B+tree leaf nodes are all linked together. To perform a range query all we need to do is:

- find an exact match search for the lowest key
- and from there, follow the linked list until we reach the leaf node with the maximum key

4. B-trees in the Context of Databases

Storing and managing data is a fundamental part of computing. Main memory is considered to be a primary data storage, but we cannot store everything in it because it's volatile and also expensive. That's why we have secondary storage that's less expensive and not volatile. Secondary storage is typically stored on magnetic disks in units called pages.

Transferring elements from disk to memory requires a read from the disk. A single disk read performs full-page access even if we are only trying to read one element from that page. Disk reads aren't as fast as main memory reads since each time we do a read there it's a seek and a rotational delay. The more disk accesses we need, the longer the search operation will take.

DBMSs leverage the logarithmic efficiency of B-tree indexing to reduce the number of reads required to find a particular record. B-trees are typically constructed so that **each node takes up a single page in memory** and they are designed to reduce the number of accesses by **requiring that each node be at least half full.**

5. Comparing Between B-trees and B+trees

Let's cover the most obvious points of comparison between B-trees and B+trees:

- In B+trees, search keys can be repeated but this is not the case for B-trees
- B+trees allow satellite data to be stored in leaf nodes only, whereas B-trees store data in both leaf and internal nodes

- In B+trees, data stored on the leaf node makes the search more efficient since we can store more keys in internal nodes – this means we need to access fewer nodes
- Deleting data from a B+tree is easier and less time consuming because we only need to remove data from leaf nodes
- Leaf nodes in a B+tree are linked together making range search operations efficient and quick

Finally, although B-trees are useful, B+trees are more popular. **In fact, 99% of database management systems use B+trees for indexing.**

This is because the B+tree holds no data in the internal nodes. This maximizes the number of keys stored in a node thereby minimizing the number of levels needed in a tree. Smaller tree depth invariably means faster search.

Algorithm Implementation B+ tree

In computer science, a **B+ tree** is a type of tree data structure. It represents sorted data in a way that allows for efficient insertion and removal of elements. It is a dynamic, multilevel index with maximum and minimum bounds on the number of keys in each node.

A B+ tree is a variation on a B-tree. In a B+ tree, in contrast to a B tree, all data are saved in the leaves. Internal nodes contain only keys and tree pointers. All leaves are at the same lowest level. Leaf nodes are also linked together as a linked list to make range queries easy.

The maximum number of keys in a record is called the order of the B+ tree.

The minimum number of keys per record is 1/2 of the maximum number of keys. For example, if the order of a B+ tree is *n*, each node (except for the root) must have between *n*/2 and *n* keys.

The number of keys that may be indexed using a B+ tree is a function of the order of the tree and its height.

For a *n*-order B+ tree with a height of *h*:

- maximum number of keys is

- minimum number of keys is

The B+ tree was first described in the paper "Rudolf Bayer, Edward M. McCreight: Organization and Maintenance of Large Ordered Indices. Acta Informatica 1: 173-189 (1972)".

Algorithm

Begin

function insert() to insert the nodes into the tree:

Initialize x as root.

if x is leaf and having space for one more info then insert a to x.

else if x is not leaf, do

Find the child of x that is going to to be traversed next.

If the child is not full, change x to point to the child.

If the child is full, split it and change x to point to one of the two parts of the child. If a is smaller

than mid key in the child, then set x as first part of the child. Else second part of the child.

End

Example Code

```
#include<iostream>
using namespace std;
struct BplusTree {
   int *d;
   BplusTree **child_ptr;
   bool l;
   int n;
}*r = NULL, *np = NULL, *x = NULL;
```

```
BplusTree* init()//to create nodes {
  int i;
  np = new BplusTree;
  np->d = new int[6];//order 6
  np->child_ptr = new BplusTree *[7];
  np->l = true;
  np->n = 0;
  for (i = 0; i < 7; i++) {
    np->child_ptr[i] = NULL;
  }
  return np;
}

void traverse(BplusTree *p)//traverse tree {
  cout<<endl;
  int i;
  for (i = 0; i < p->n; i++) {
    if (p->l == false) {
      traverse(p->child_ptr[i]);
    }
    cout << " " << p->d[i];
  }
  if (p->l == false) {
    traverse(p->child_ptr[i]);
  }
  cout<<endl;
}

void sort(int *p, int n)//sort the tree {
```

```
  int i, j, t;
  for (i = 0; i < n; i++) {
    for (j = i; j <= n; j++) {
      if (p[i] >p[j]) {
        t = p[i];
        p[i] = p[j];
        p[j] = t;
      }
    }
  }
}

int split_child(BplusTree *x, int i) {
  int j, mid;
  BplusTree *np1, *np3, *y;
  np3 = init();
  np3->l = true;
  if (i == -1) {
    mid = x->d[2];
    x->d[2] = 0;
    x->n--;
    np1 = init();
    np1->l = false;
    x->l = true;
    for (j = 3; j < 6; j++) {
      np3->d[j - 3] = x->d[j];
      np3->child_ptr[j - 3] = x->child_ptr[j];
      np3->n++;
      x->d[j] = 0;
```

```
      x->n--;
    }
    for (j = 0; j < 6; j++) {
      x->child_ptr[j] = NULL;
    }
    np1->d[0] = mid;
    np1->child_ptr[np1->n] = x;
    np1->child_ptr[np1->n + 1] = np3;
    np1->n++;
    r = np1;
  } else {
    y = x->child_ptr[i];
    mid = y->d[2];
    y->d[2] = 0;
    y->n--;
    for (j = 3; j <6 ; j++) {
      np3->d[j - 3] = y->d[j];
      np3->n++;
      y->d[j] = 0;
      y->n--;
    }
    x->child_ptr[i + 1] = y;
    x->child_ptr[i + 1] = np3;
  }
  return mid;
}

void insert(int a) {
  int i, t;
```

```
  x = r;
  if (x == NULL) {
    r = init();
    x = r;
  } else {
    if (x->l== true && x->n == 6) {
      t = split_child(x, -1);
      x = r;
      for (i = 0; i < (x->n); i++) {
        if ((a >x->d[i]) && (a < x->d[i + 1])) {
        i++;
        break;
      } else if (a < x->d[0]) {
        break;
      } else {
        continue;
      }
    }
    x = x->child_ptr[i];
  } else {
    while (x->l == false) {
      for (i = 0; i < (x->n); i++) {
        if ((a >x->d[i]) && (a < x->d[i + 1])) {
          i++;
          break;
        } else if (a < x->d[0]) {
          break;
        } else {
          continue;
```

```
        }
      }
      if ((x->child_ptr[i])->n == 6) {
        t = split_child(x, i);
        x->d[x->n] = t;
        x->n++;
        continue;
      } else {
        x = x->child_ptr[i];
      }
    }
  }
}
  x->d[x->n] = a;
  sort(x->d, x->n);
  x->n++;
}

int main() {
  int i, n, t;
  cout<<"enter the no of elements to be inserted\n";
  cin>>n;
  for(i = 0; i < n; i++) {
    cout<<"enter the element\n";
    cin>>t;
    insert(t);
  }
  cout<<"traversal of constructed B tree\n";
  traverse(r);
```

```
}
```

Output

```
enter the no of elements to be inserted
10
enter the element
10
enter the element
20
enter the element
30
enter the element
40
enter the element
50
enter the element
60
enter the element
70
enter the element
80
enter the element
90
enter the element
100
traversal of constructed B tree
10 20
30
```

40 50

60

70 80 90 100