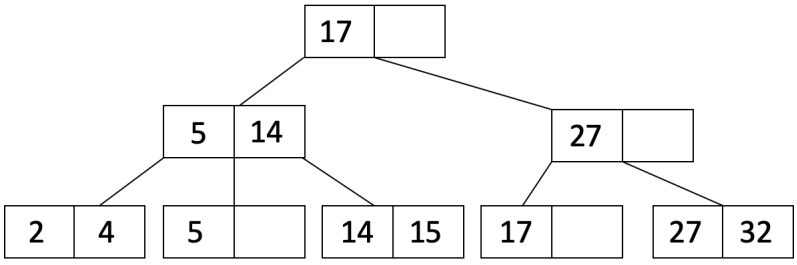
CS W186
Spring 2020

B+ Trees

1 Introduction

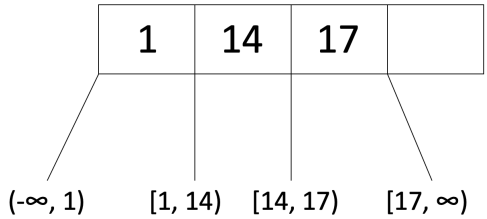
An index is a data structure that helps speed up reads on a specific key. We use indexes to make queries run faster, especially queries that are run frequently. For example, to make login faster you may want to build an index on username so that you can quickly find the row of the user that is trying to log in. In this course note, we will learn about B+ trees which is a specific type of index. Here is an example of what a B+ tree looks like:



2 Properties

- The number d is the order of a B+ tree. Each node (with the exception of the root node) must have $d \leq x \leq 2d$ entries assuming no deletes happen (it's possible for leaf nodes to end up with $<$ d entries if you delete data). The entries of the node must be **sorted**.
- In between each entry of an inner node, there is a pointer to a child node. Since there are at most $2d$ entries in a node, the node may have at most $2d + 1$ child pointers. This is also called the tree's fanout.
- The keys in the children to the left of an entry must be less than the entry while the keys in the children to the right must be greater than or equal to the entry.

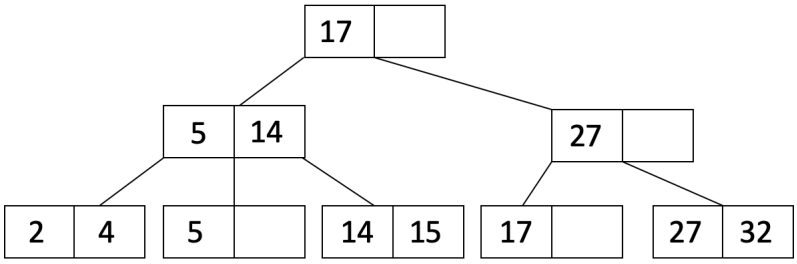
For example, here is a node of an order $d = 2$ tree:



Note that the node satisfies the order requirement ($d \leq x \leq 2d$) because $d = 2$ and this node has 3 entries which satisfies $2 \leq x \leq 4$.

- Because of the sorted and children property, we can traverse the tree down to the leaf to find our desired record. This is similar to BSTs (Binary Search Trees).
- Every root to leaf path has the same number of **edges** - this is the height of the tree. In this sense, B+ trees are **always** balanced.
- Only the leaf nodes contain records (or pointers to records - this will be explained later). The inner nodes (which are the non-leaf nodes) do not contain the actual records.

For example, here is an order d=1 tree:



3 Insertion

To insert an entry into the B+ tree, follow this procedure:

(1) Find the leaf node L in which you will insert your value. You can do this by traversing down the tree. Add the key and the record to the leaf node in order.

(2) If L overflows (L has more than $2d$ entries)...

  (a) Split into L_1 and L_2. Keep d entries in L_1 (this means $d + 1$ entries will go in L_2).

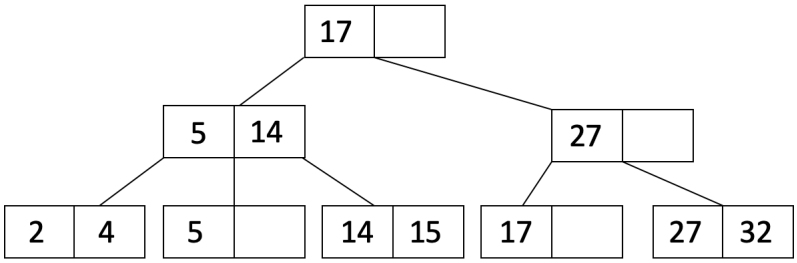
  (b) If L was a leaf node, **COPY** L_2's first entry into the parent. If L was not a leaf node, **MOVE** L_2's first entry into the parent.

  (c) Adjust pointers.

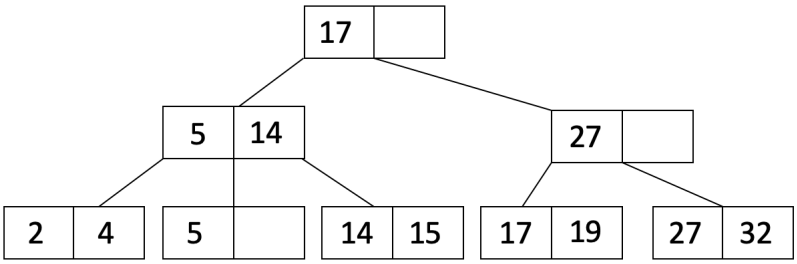
(3) If the parent overflows, then recurse on it by doing step 2 on the parent.

Note: we want to **COPY** leaf node data into the parent so that we don't lose the data in the leaf node. Remember that every key that is in the table that the index is built on must be in the leaf nodes! Being in a inner node does not mean that key is actually still in the table. On the other hand, we can **MOVE** inner node data into parent nodes because the inner node does not contain the actual data, they are just a reference of which way to search when traversing the tree.

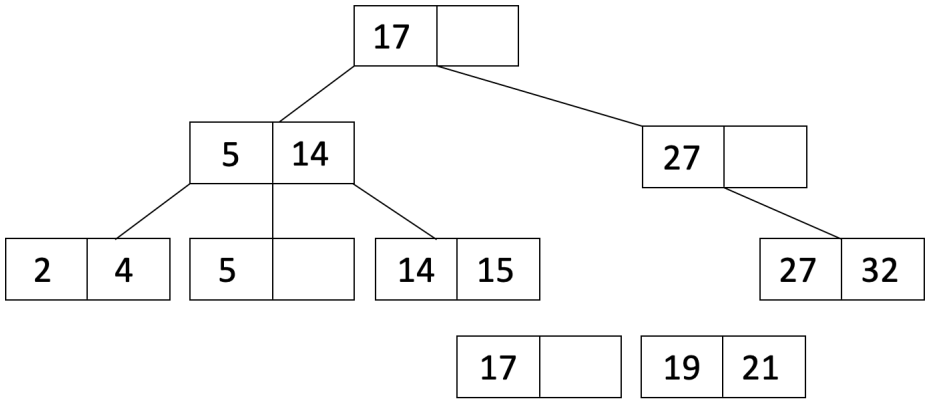
Let's take a look at an example to better understand this procedure! We start with the following order $d = 1$ tree:



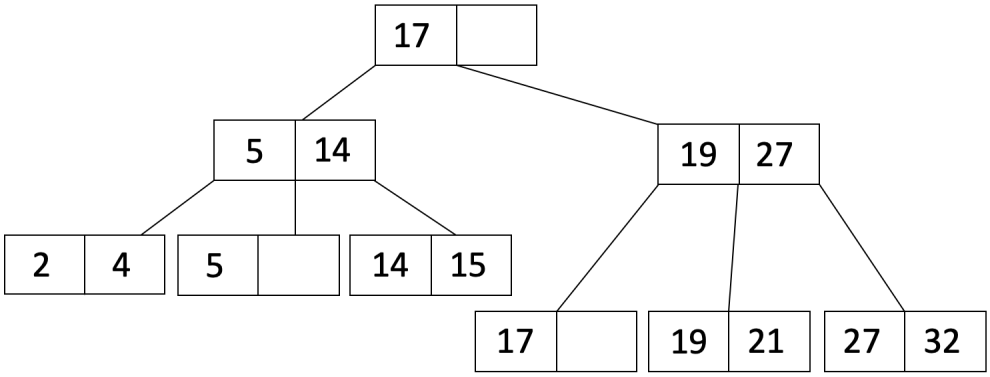
Let's insert 19 into our tree. When we insert 19, we see that there is space in leaf node with 17:



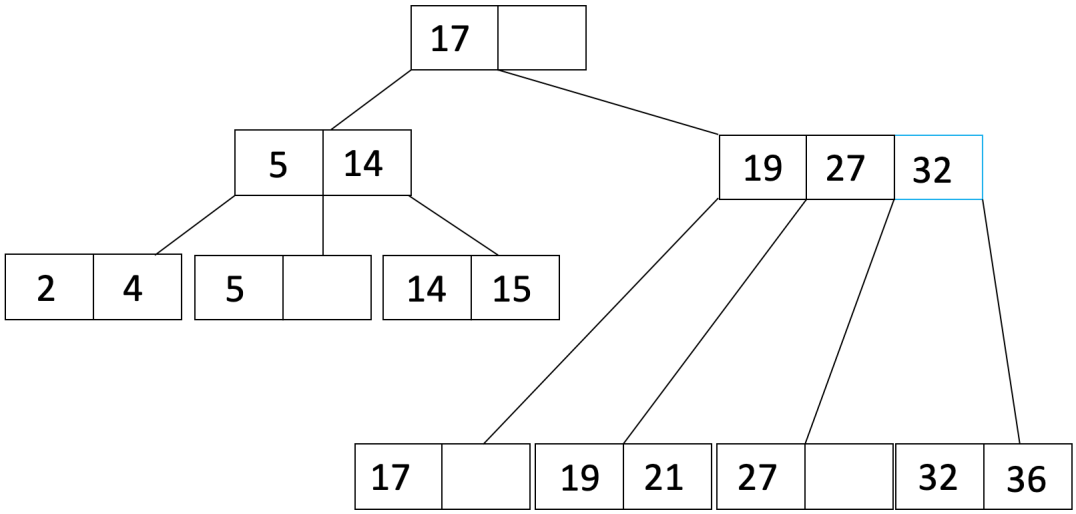
Now let's insert 21 into our tree. When we insert 21, it causes one of the leaf nodes to overflow. Therefore, we split this leaf node into two leaf nodes as shown below:



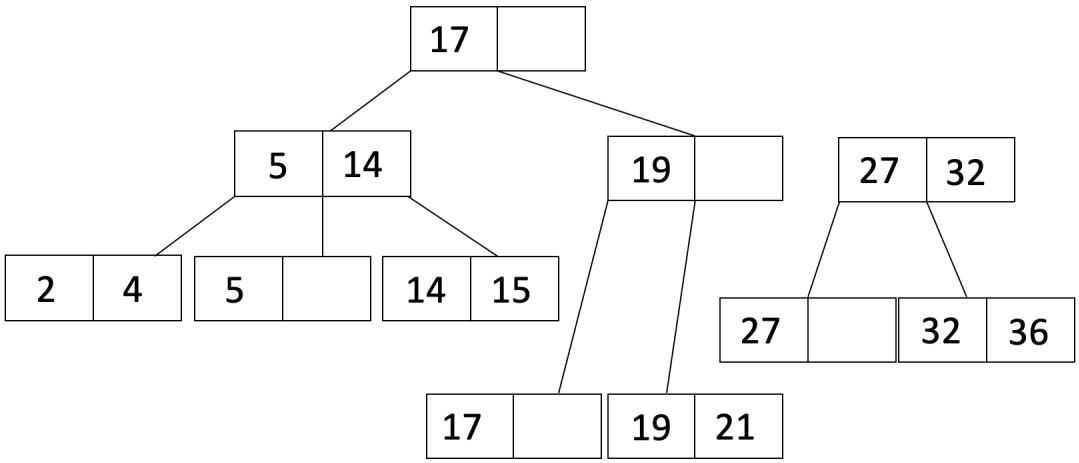
Since we split a leaf node, we will **COPY** L_2's first entry up to the parent and adjust pointers to get:



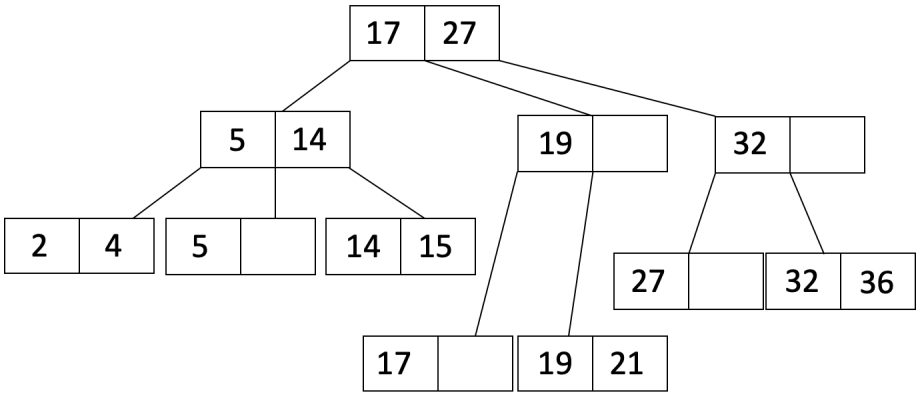
Let's do one more insertion. This time we will insert 36. When we insert 36, the leaf overflows so we will do the same procedure as when we inserted 21 to get:



Notice that now the parent node overflowed, so now we must recurse. We will split the parent node to get:



Since it was an inner node that overflowed, we will **MOVE** L_2's first entry up to the parent and adjust pointers to get:



4 Deletion

To delete a value, just find the appropriate leaf and delete the unwanted value from that leaf. That's all there is to it. (Yes, technically we could end up violating some of the invariants of a B+ tree. That's okay because in practice we get *way* more insertions than deletions so something will quickly replace whatever we delete.)

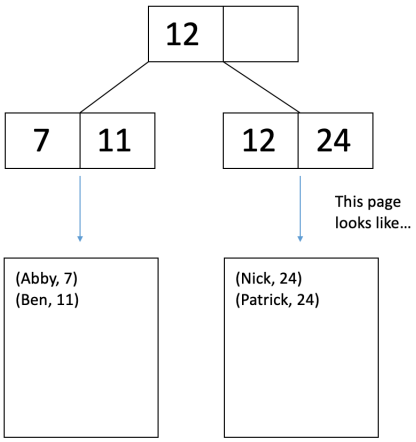
Reminder: We never delete inner node keys because they are only there for search and not to hold data.

5 Storing Records

Up until now, we have not discussed how the records are actually stored in the leaves. Let's take a look at that now. There are three ways of storing records in leaves:

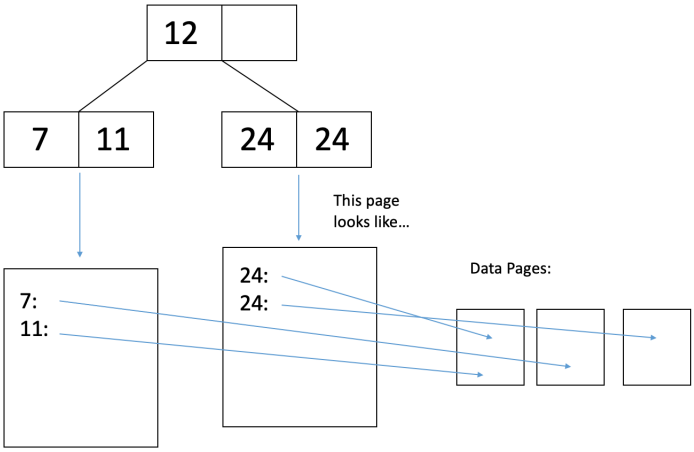
- **Alternative 1**

  In the Alternative 1 scheme, the leaf pages are the data pages. Rather than containing pointers to records, the leaf pages contain the records themselves.



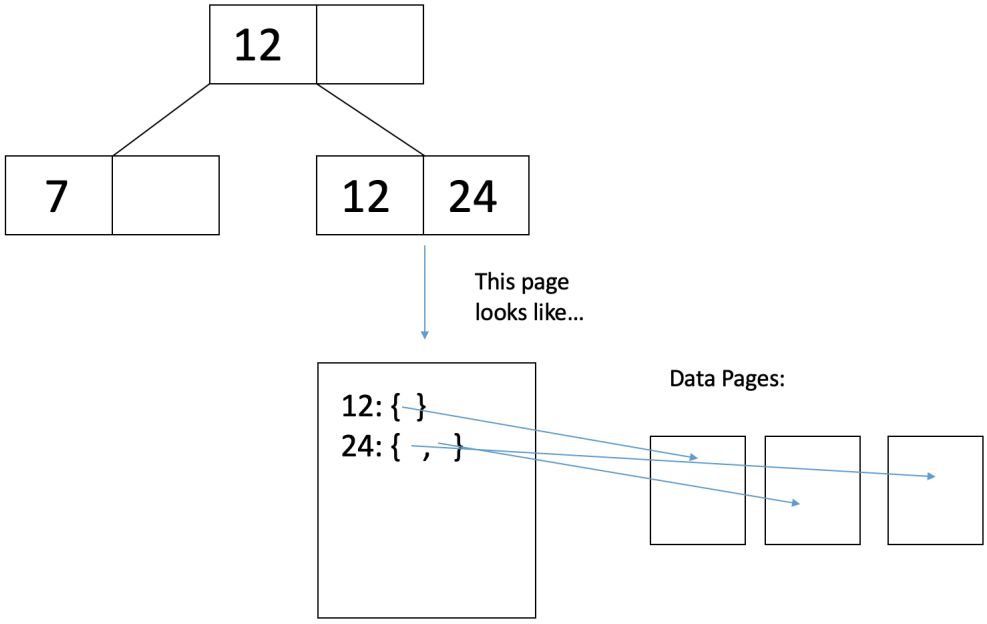
- **Alternative 2**

  In the Alternative 2 scheme, the leaf pages hold pointers to the corresponding records.



- **Alternative 3**

  In the Alternative 3 scheme, the leaf pages hold linked lists of pointers to the corresponding records.

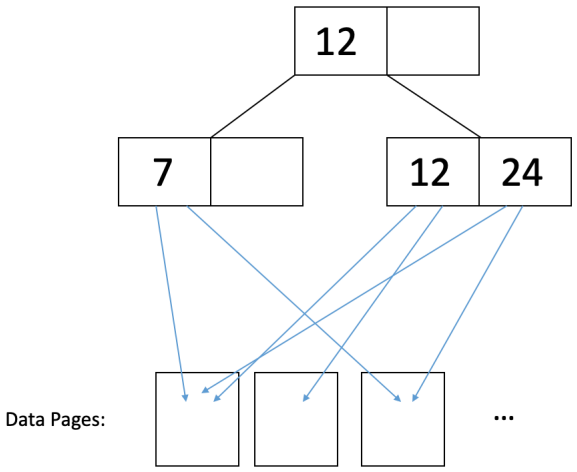


6 Clustering

Now that we've discussed how records are stored in the leaf nodes, we will also discuss how data on the data pages are organized. Clustered/unclustered refers to how the data pages are structured. Unclustering only applies to Alternative 2 or 3.

- **Unclustered**

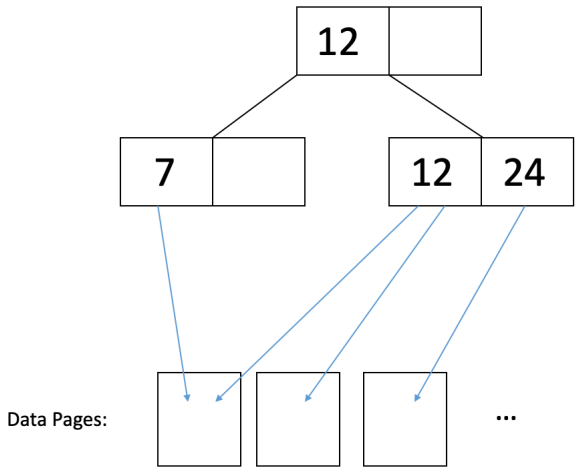
  In an unclustered index, the data pages are complete chaos. Thus, odds are that you're going to have to read a separate page for each of the records you want. For instance, consider this illustration:



  In the figure above, if we want to read records with 12 and 24, then we would have to read in each of the data pages they point to in order to retrieve all the records associated with these keys.

- **Clustered**

  In a clustered index, the data pages are sorted on the same index on which you've built your B+ tree. This does not mean that the data pages are sorted exactly, just that keys are roughly in the same order as data. The difference in I/O cost therefore comes from caching, where two records with close keys will likely be in the same page, so the second one can be read from the cached page. Thus, you typically just need to read one page to get all the records that have a common / similar key. For instance, consider this illustration:



  In the figure above, we can read records with 7 and 12 by reading 2 pages. If we do sequential reads of the leaf node values, the data page is largely the same. In conclusion,

- UNCLUSTERED = $\sim$ 1 I/O per record.
- CLUSTERED = $\sim$ 1 I/O per page of records.

7 Counting IO's

Here's the general procedure. It's a good thing to write on your cheat sheet:

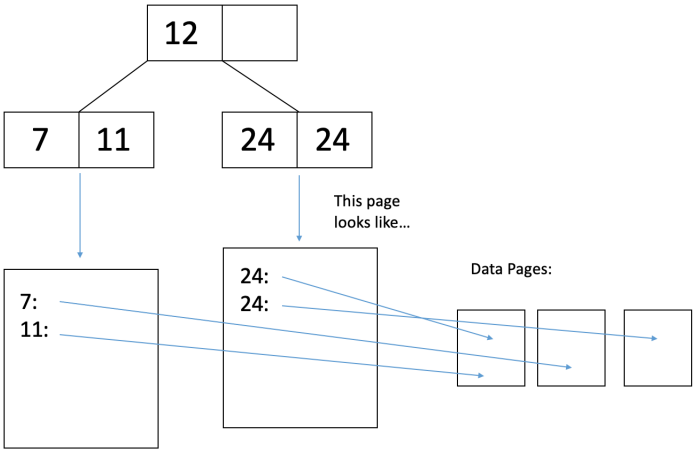
(1) Read the appropriate root-to-leaf path.

(2) Read the appropriate data page(s). If we need to read multiple pages, we will allot a read IO for each page. In addition, we account for clustering for Alt. 2 or 3 (see below.)

(3) Write data page, if you want to modify it. Again, if we want to do a write that spans multiple data pages, we will need to allot a write IO for each page.

(4) Update index page(s).

Let's look at an example. See the following B+ tree:



We want to delete the only 11-year-old from our database. How many I/O's will it take?

- One I/O for each of the 2 relevant index pages (an index page is an inner node or a leaf node).
- One I/O to read the data page where the 11-year-old's record is. Once it's in RAM we can delete the record from the page.
- One I/O to write the modified data page back to disk.
- Now that there are no more 11-year-olds in our database we should remove the key "11" from the leaf page of our B+ tree, which we already read in Step 1. We do so, and then it takes one I/O to write the modified leaf page to disk.

8 Bulk Loading

The insertion procedure we discussed before is great for making additions to an existing B+ tree. If we want to construct a B+ from scratch, however, we can do better. This is because if we use the insertion procedure we would have to traverse the tree each time we want to insert something new. Instead, we will use **bulkloading**:

(1) Sort the data on the key the index will be built on.

(2) Fill leaf pages until some fill factor f.

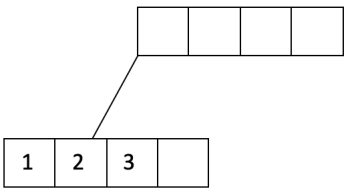
(3) Add pointer from parent to leaf page. If the parent overflows, we will follow a procedure similar to insertion. We will split the parent into two nodes:

   (a) Keep d entries in L_1 (this means $d+1$ entries will go in L_2).

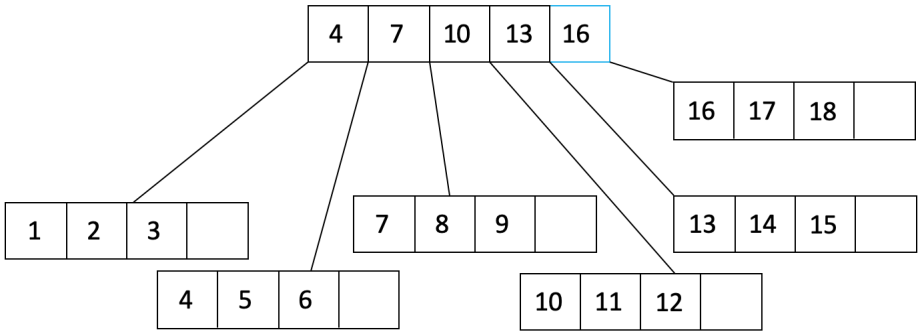
   (b) Since a parent node overflowed, we will **MOVE** L_2's first entry into the parent.

(4) Adjust pointers.

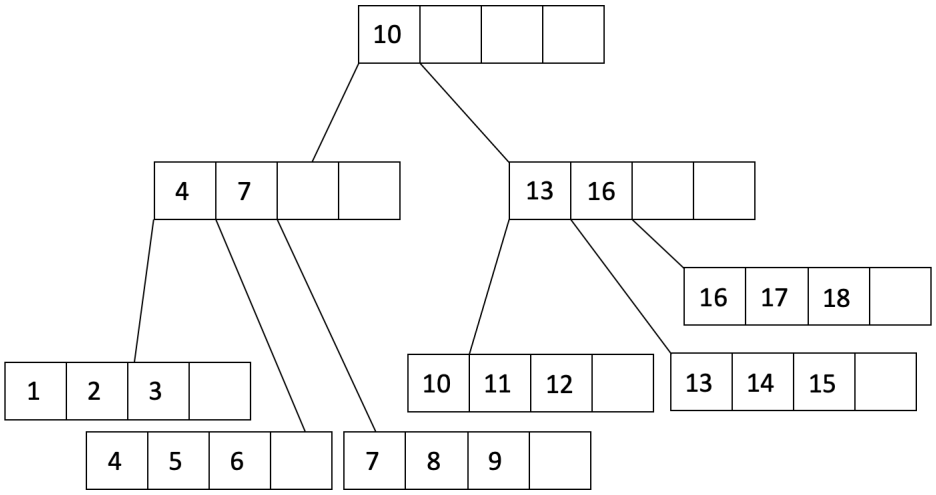
Let's look at an example. Let's say our fill factor is $\frac{3}{4}$ and we want to insert $1, \ldots, 20$ into an order $d = 2$ tree. We will start by filling a leaf page until our fill factor:



We have filled a leaf node to the fill factor of $\frac{3}{4}$ and added a pointer from the parent node to the leaf node. Let's continue filling:

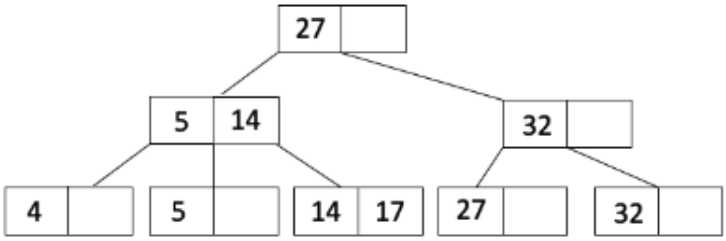


In the figure above, we see that the parent node has overflowed. We will split the parent node into two nodes and create a new parent:



9 Practice Questions

1. What is the maximum number of entries we can add to the following B+ tree?



2. What is the minimum number of entries we can add to the B+ tree above which will cause the height to increase?

3. How many I/O's would it cost to insert an entry into our table if we had a height 2, unclustered alternative 3 B+ tree in the worst case? Assume that the cache is empty at the beginning and there are at most 10 entries with the same key. Assume each page is at most $\frac{2}{3}$ full.

10 Solutions

1. The maximum number of entries we can add is the total capacity of a height 2, order $d = 1$ tree minus the current number of entries. The total capacity is $(2d)(2d+1)^h = (2)(3^2) = 18$. The current number of entries is 6 because the entries are in the leaf nodes. Therefore, we can add a maximum of $18 - 6 = 12$.

2. The minimum number of entries we can add to cause the height to increase will be 3.

   We will add 20 to the fullest leaf node which is $14, 17$ which will cause it to split into 14 and $17, 20$. Then we copy 17 to its parent $5, 14$, which causes it to split into 5 and 17 and 14 will be pushed up so the root will have $14, 27$.

   Then we will insert 21 into the fullest leaf which will now be $17, 20$, which will cause it to split into 17 and $20, 21$. Then we copy 20 up to its parent which will become $17, 20$.

   Then we can insert 22 into the fullest leaf $20, 21$ which will split into 20 and $21, 22$. Then we copy 21 up to the parent $17, 20$, which will cause it to split into 17 and 21 because 20 will be pushed up to its parent (the root). This root will also split because it already has $14, 27$. When the root splits, we know that the height has increased.

3. First, we need to check if our table already contains this entry. We will search down the B+ tree for the key which will cost us 3 I/O's because the tree is height 2. Then we get to a leaf node and we see that there are at most 10 entries with the same key. We need to check each of these entries to make sure it's not the same as our current entry which is 10 I/O's because our B+ tree is unclustered. In the worst case, none of the 10 entries match the entry we want to insert so we will have to go ahead and add it to the table. We will add the entry to any page that has space (we already have this page in our cache from the previous part so this does not incur an I/O) and then writing this page back to disk which will cost 1 I/O. We also have to update our B+ tree leaf node to include a pointer to this new entry so we will have to write the node back to disk which is 1 I/O. Therefore, we will have a total of $3 + 10 + 1 + 1 = 15$ I/O's.