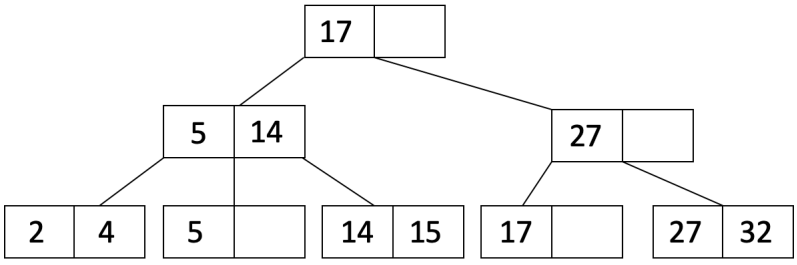
CS W186
Fall 2019

B+ Trees

1 Introduction

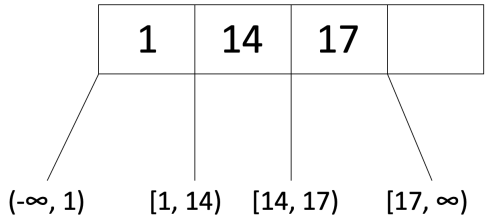
An index is a data structure that helps speed up reads on a specific key. A use case where you might want to use an index is finding where a specific topic is discussed in a textbook. In this course notes, we will learn about B+ trees which is a specific type of index. Here is an example of what a B+ 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.

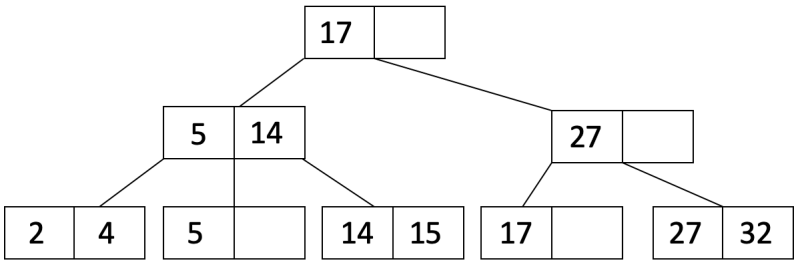
For example, here is a node of an order $d = 2$ tree:



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+ 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.

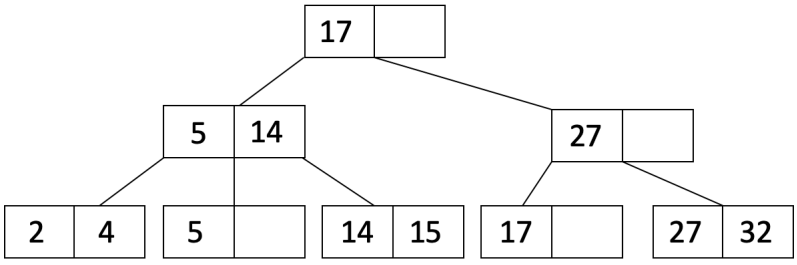
  (c) Adjust pointers.

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

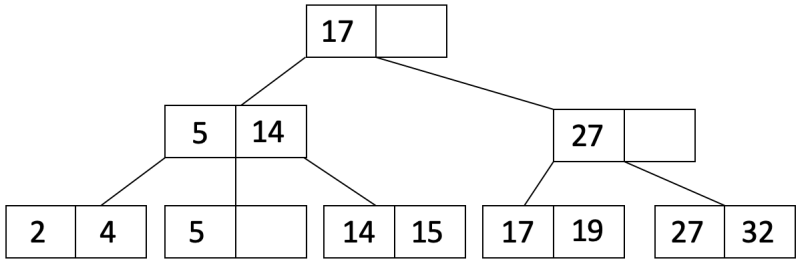
Note: we want to **COPY** leaf node data into the parent so that we don't lose the data in the leaf node. On the other hand, we can **MOVE** inner node data into parent nodes because the inner node does not contain the actual data, they are a reference of which way to search when traversing the tree.

B+ Trees

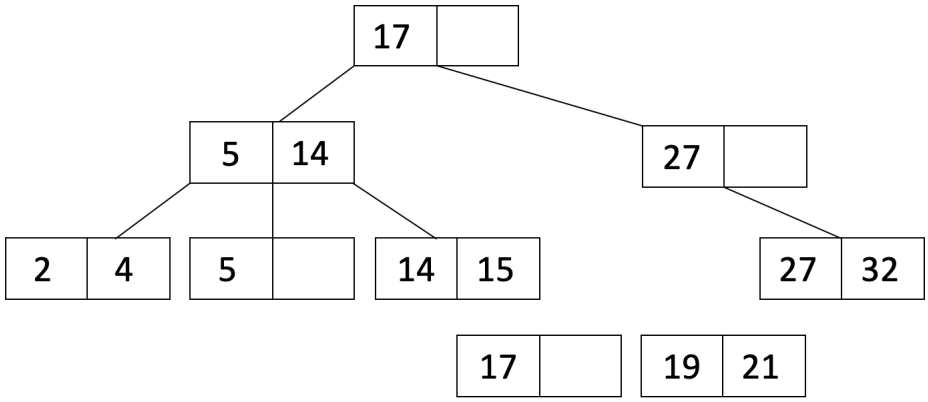
Let's take a look an example to better understand this procedure! We start with the following order $d = 1$ tree:



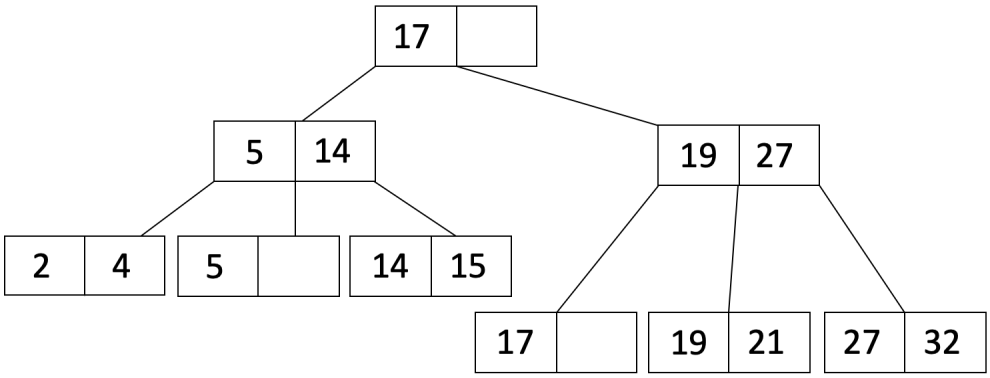
Let's insert 19 into our tree. When we insert 19, we see that there is space in leaf node with 17:



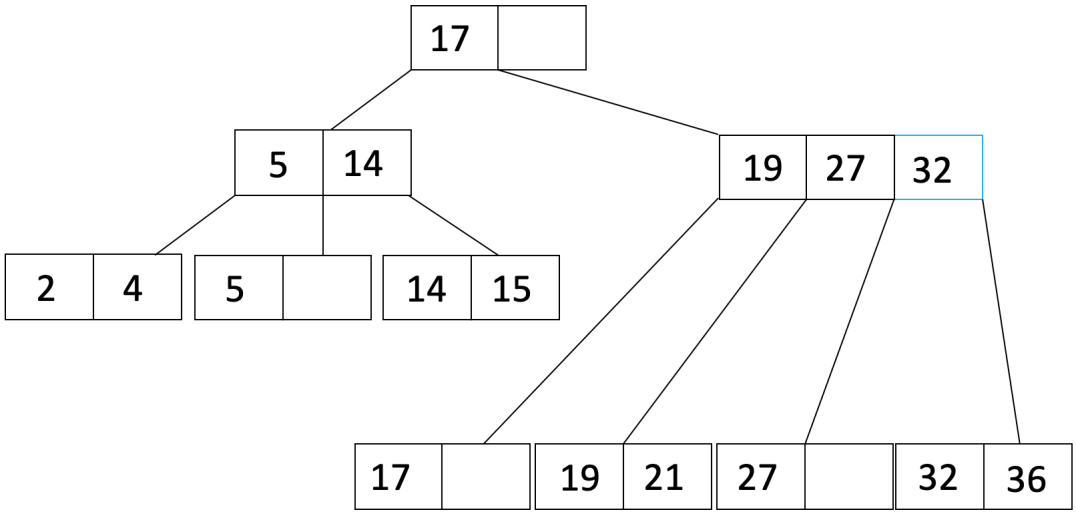
Now let's insert 21 into our tree. When we insert 21, it causes one of the leaf nodes to overflow. Therefore, we split this leaf node into two leaf nodes as shown below:



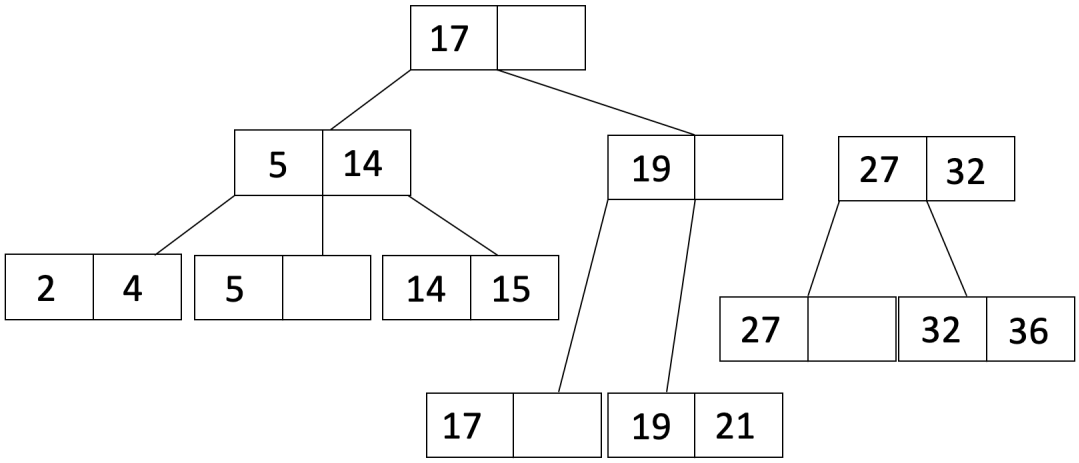
Since we split a leaf node, we will **COPY** L_2's first entry up to the parent and adjust pointers to get:



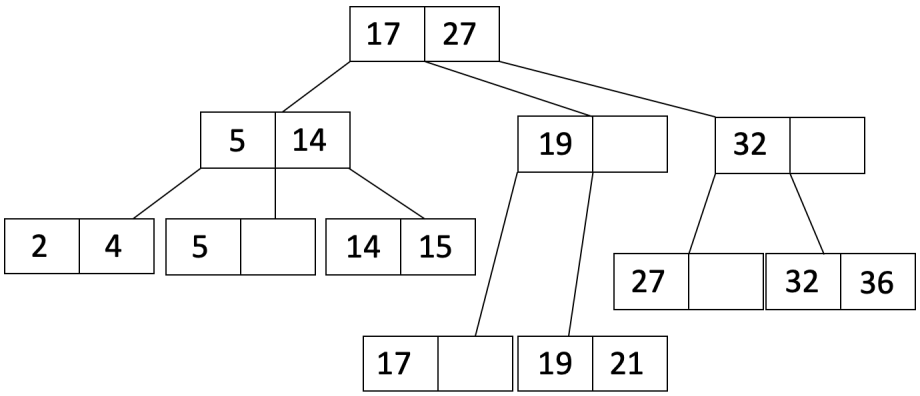
Let's do one more insertion. This time we will insert 36. When we insert 36, the leaf overflows so we will do the same procedure as when we inserted 21 to get:



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.)

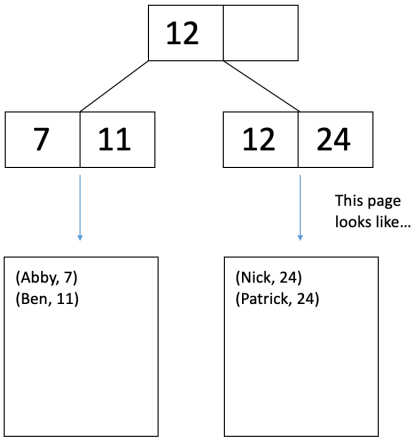
Reminder: We never delete inner node keys because they are only there for search and not to hold data.

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:

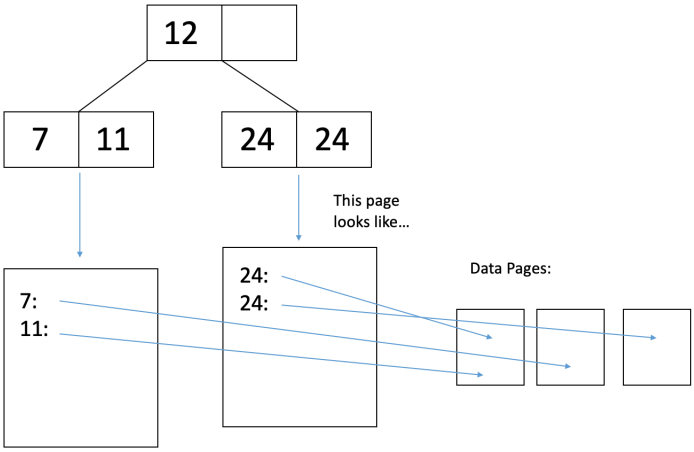
- **Alternative 1**

  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.



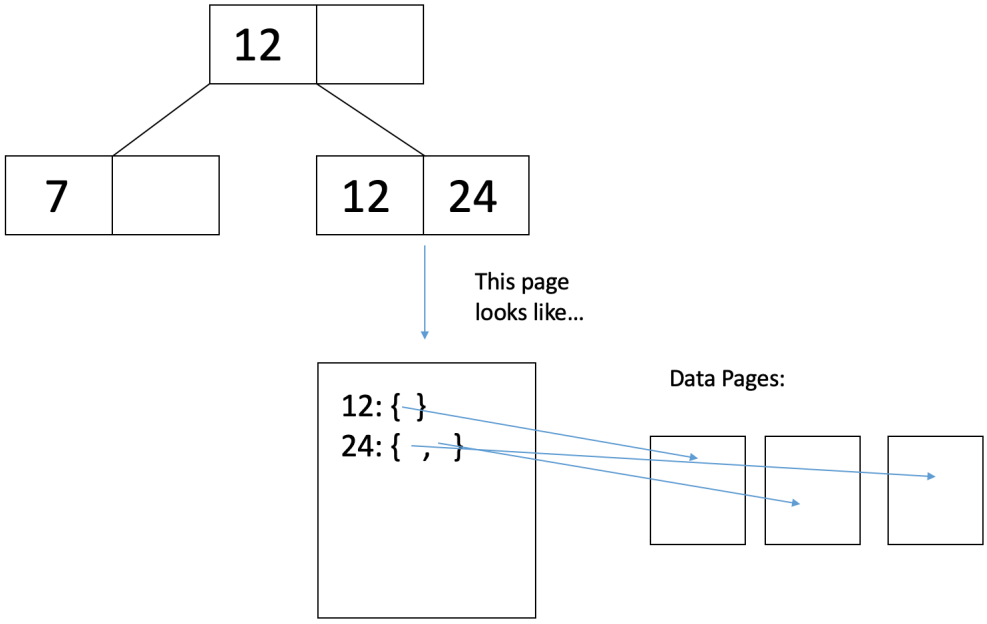
- **Alternative 2**

  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.

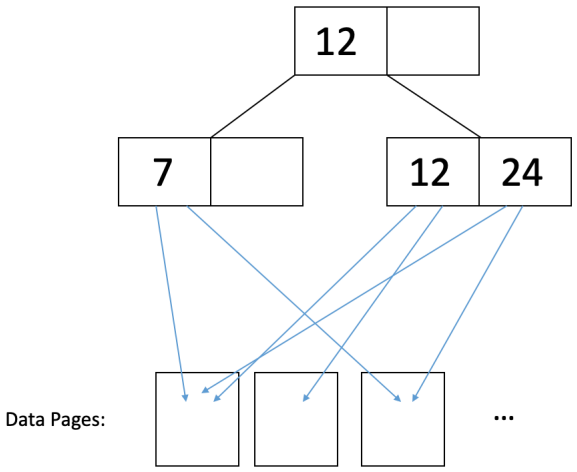


6 Clustering

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**

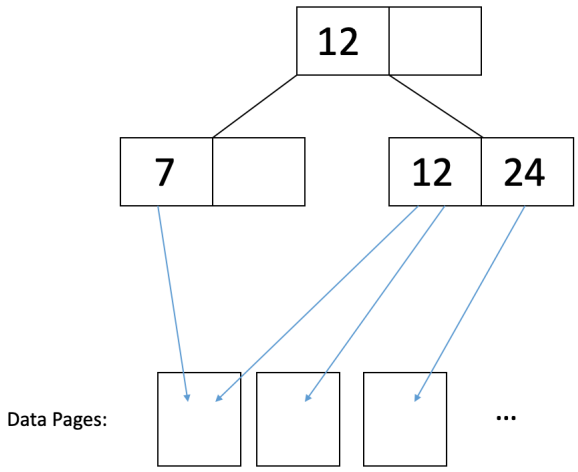
  In an unclustered index, the data pages are complete chaos. Thus, odds are that you're going to have to read a separate page for each of the records you want. For instance, consider this illustration:



  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:

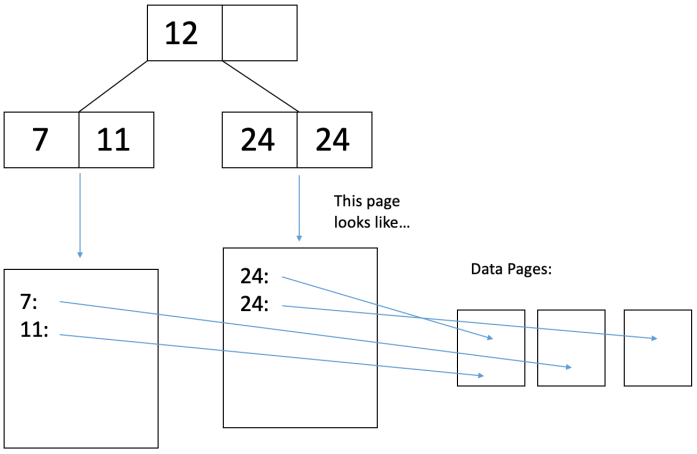
(1) Read the appropriate root-to-leaf path.

(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

What if we want to insert a bunch of data already in sorted order? If we use the insertion procedure we discussed earlier in this course note, we would have to traverse the tree each time we want to insert something new. This is not efficient since we already know that our input is in sorted order. Therefore, we can use a bulk loading procedure:

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

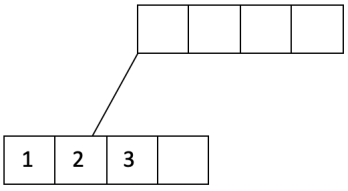
(2) Add pointer from parent to leaf page. If the parent overflows, we will follow a procedure similar to insertion. We will split the parent into two nodes:

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

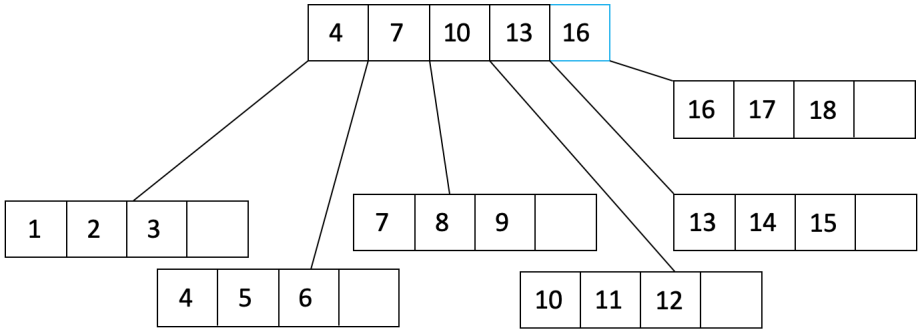
   (b) Since a parent node overflowed, we will **MOVE** L_2's first entry into the parent.

(3) 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: