Trees can be used to store entire records from a database, serving as an in-memory representation of the collection of records in a file.
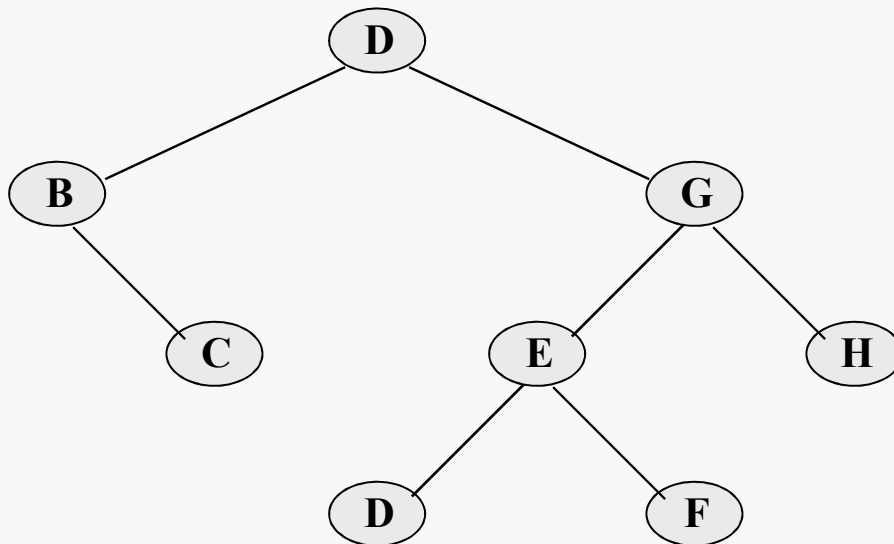
Trees can also be used to store indices of the collection of records in a file.

In either case, if the collection of records is quite large, the tree may be so large that it is unacceptable to store it all in memory at once.

For example, if we have a database file holding 2^{30} records, and each index entry requires 8 bytes of storage, a BST holding the index would require 2^{30} nodes, each taking 16 bytes of memory (assuming 32-bit pointers), or 16 GB of memory.

An alternative would be to store the entire tree in a file on disk, and only load the immediately relevant portions of it into memory…

It is a relatively simple matter to write any binary tree to a disk file, by representing each tree node by a data record that holds the data element and two file offsets specifying the locations of the children, if any of that node.

	Data	lChild	rChild
0	D	24	72
24	B	-1	48
48	C	-1	-1
72	G	96	168
96	E	120	144
120	D	-1	-1
144	F	-1	-1
168	H	-1	-1

The nodes don't need to be stored in any particular order.

Null pointers may be represented by a negative offset.

The problem is that this disk representation will require too many individual disk accesses when processing a typical tree operation, such as a search or a traversal.

Why?

These tree operations typically require transiting from a node to one or both of its children.

But there's no reason that the child nodes will be stored anywhere near the parent node (although we could at least guarantee that siblings are adjacent).

Since each node stores only one data value, and the nodes we might well perform one disk access for every node that is accessed during the tree operation.

Given the extremely slow nature of disk access, this is unacceptable.

	Data	lChild	rChild
0	D	24	72
24	B	-1	48
48	C	-1	-1
72	G	96	168
96	E	120	144
120	D	-1	-1
144	F	-1	-1
168	H	-1	-1

A B-tree of order *m* is a multi-way tree such that:

- the root has at least two subtrees, unless it is a leaf

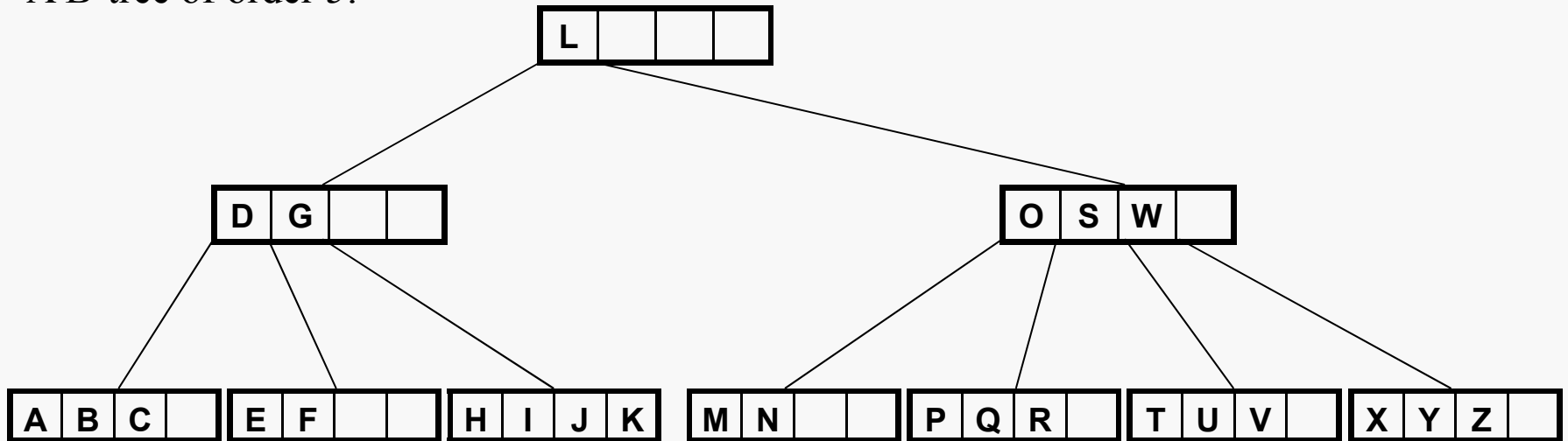- each nonroot and nonleaf node holds *k* – *1* data values, and *k* pointers to subtrees, where

$$\lceil m/2 \rceil \le k \le m$$

- each leaf node holds *k* – *1* data values, where $\lceil m/2 \rceil \le k \le m$

- all leaves are on the same level

- the data values in each node are in ascending order

- for all *i*, the data values in the first *i* children are less than the *i*-th data value

- for all *i*, the data values in the last *m* – *i* children are larger than the *i*-th data value

So, a B-tree is generally at least half full, has a relatively small number of levels, and is perfectly balanced. Typically, *m* will be fairly large.

A B-tree of order 5:

```
                              ┌─┬─┬─┬─┐
                              │L│ │ │ │
                              └─┴─┴─┴─┘
                  ┌──────────────┘        └──────────────────┐
            ┌─┬─┬─┬─┐                              ┌─┬─┬─┬─┐
            │D│G│ │ │                              │O│S│W│ │
            └─┴─┴─┴─┘                              └─┴─┴─┴─┘
       ┌─────┼─────┐                       ┌────┬────┼────┬────┐
 ┌─┬─┬─┬─┐ ┌─┬─┬─┬─┐ ┌─┬─┬─┬─┐   ┌─┬─┬─┬─┐ ┌─┬─┬─┬─┐ ┌─┬─┬─┬─┐ ┌─┬─┬─┬─┐
 │A│B│C│ │ │E│F│ │ │ │H│I│J│K│   │M│N│ │ │ │P│Q│R│ │ │T│U│V│ │ │X│Y│Z│ │
 └─┴─┴─┴─┘ └─┴─┴─┴─┘ └─┴─┴─┴─┘   └─┴─┴─┴─┘ └─┴─┴─┴─┘ └─┴─┴─┴─┘ └─┴─┴─┴─┘
```

Since a binary search may be applied to the data values in each node, searching is highly efficient.

Insertion follows similar logic to the BST, with the complications that we must search the list of values in each node, and make nodes obey the more complex restriction
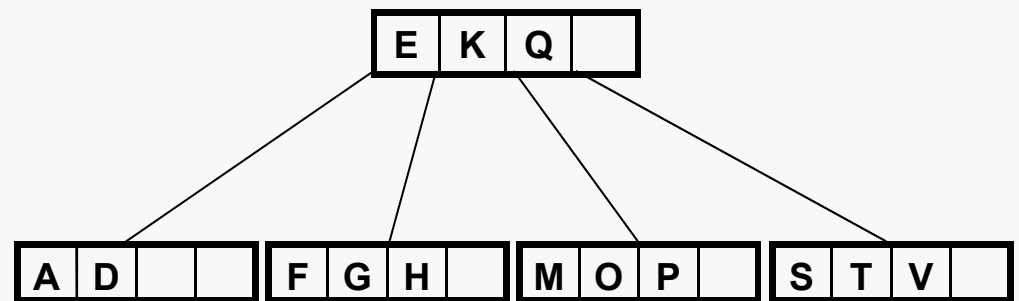
$$\lceil m/2 \rceil \le k \le m$$

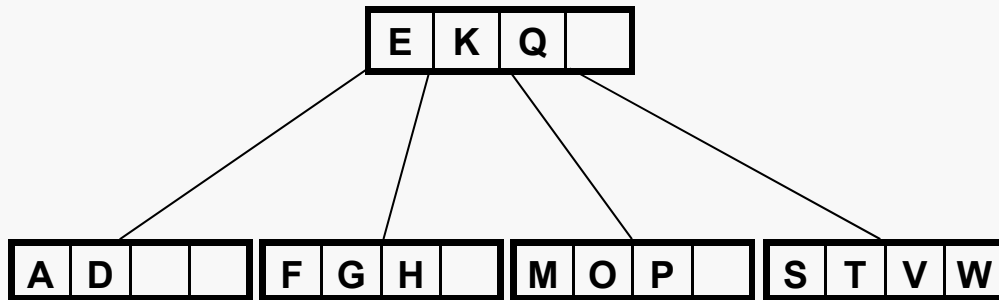where k is the number of children the node has.

The basic idea is the same: search for the appropriate leaf, add the new value, then split and promote as necessary.

For instance, inserting the values **W** and then **X** into the B tree at right would cause the right-most leaf to split and the value **V** to be promoted to the root.

Then, inserting the value **Z** would cause the root to split, and the value **Q** to be promoted to a new root node.
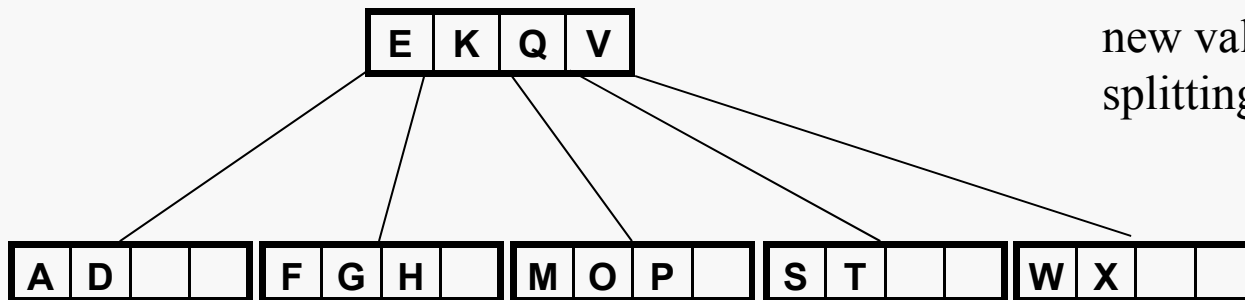
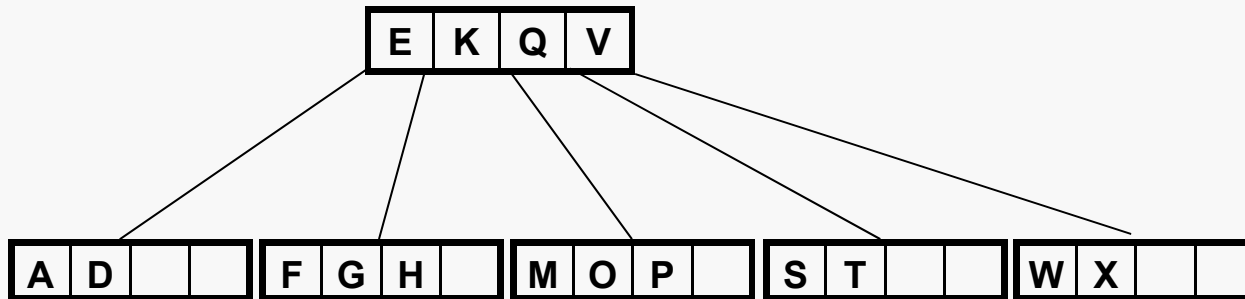| E	K	Q	

| A | D | | | | F | G | H | | | M | O | P | | | S | T | V | |

```
          ┌───┬───┬───┬───┐
          │ E │ K │ Q │   │
          └───┴───┴───┴───┘

┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐
│ A │ D │   │   │  │ F │ G │ H │   │  │ M │ O │ P │   │  │ S │ T │ V │ W │
└───┴───┴───┴───┘  └───┴───┴───┴───┘  └───┴───┴───┴───┘  └───┴───┴───┴───┘
```

Inserting **W** just fills the leaf.

```
              ┌───┬───┬───┬───┐
              │ E │ K │ Q │ V │
              └───┴───┴───┴───┘

┌───┬───┬───┬───┐ ┌───┬───┬───┬───┐ ┌───┬───┬───┬───┐ ┌───┬───┬───┬───┐ ┌───┬───┬───┬───┐
│ A │ D │   │   │ │ F │ G │ H │   │ │ M │ O │ P │   │ │ S │ T │   │   │ │ W │ X │   │   │
└───┴───┴───┴───┘ └───┴───┴───┴───┘ └───┴───┴───┴───┘ └───┴───┴───┴───┘ └───┴───┴───┴───┘
```

Inserting **X** causes the leaf to overflow. So, we split the leaf and promote the median value, which is **V**, up one level.

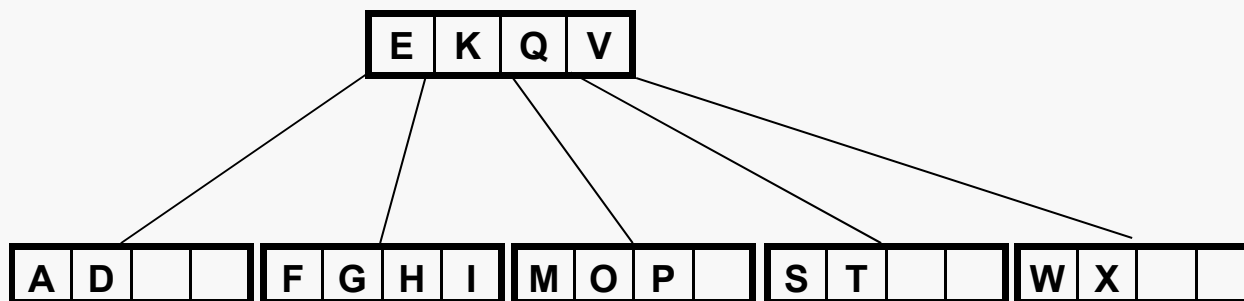The node there had room for the new value, so no further splitting occurs.

```
        ┌─┬─┬─┬─┐
        │E│K│Q│V│
        └─┴─┴─┴─┘
```

```
┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐
│A│D│ │ │  │F│G│H│ │  │M│O│P│ │  │S│T│ │ │  │W│X│ │ │
└─┴─┴─┴─┘  └─┴─┴─┴─┘  └─┴─┴─┴─┘  └─┴─┴─┴─┘  └─┴─┴─┴─┘
```

Inserting **I** just fills the second leaf.

Then, inserting **J** causes the leaf to overflow. So, we split the leaf and promote the median value, which is **H**, up one level.
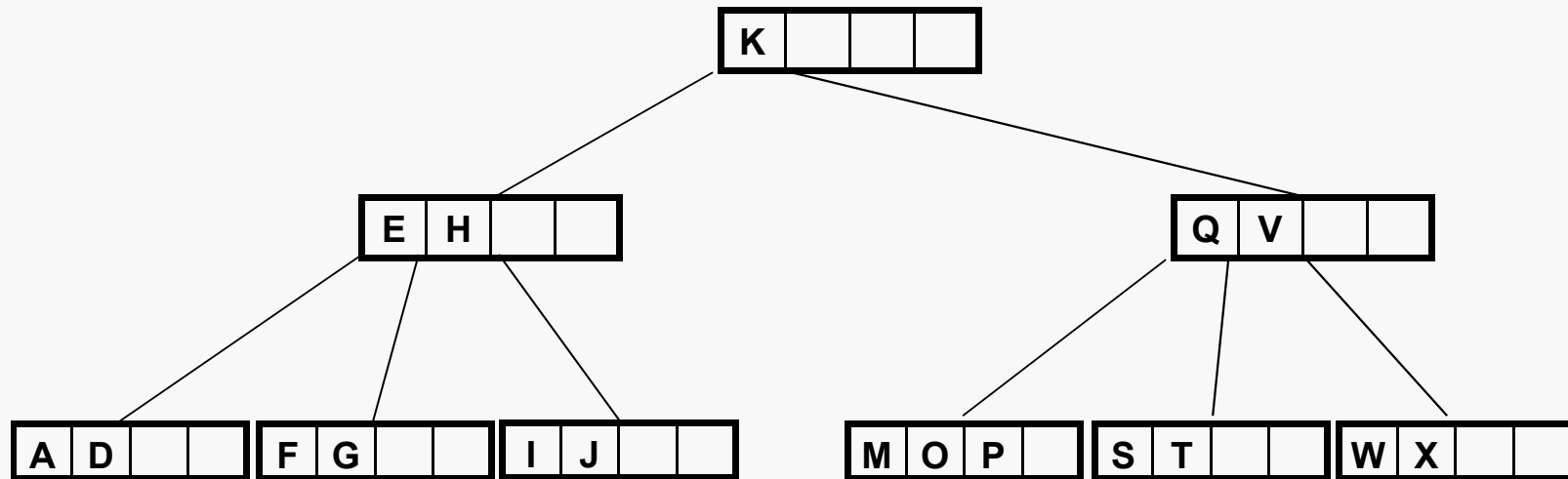
Now, the parent is full, and so splitting proceeds…

```
        ┌─┬─┬─┬─┐
        │E│K│Q│V│
        └─┴─┴─┴─┘
```

```
┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐ ┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐  ┌─┬─┬─┬─┐
│A│D│ │ │  │F│G│H│I│ │M│O│P│ │  │S│T│ │ │  │W│X│ │ │
└─┴─┴─┴─┘  └─┴─┴─┴─┘ └─┴─┴─┴─┘  └─┴─┴─┴─┘  └─┴─┴─┴─┘
```

Data Structures & Algorithms

Splitting the root sends **K** up:

So, the B-tree grows by pushing up a new root, which keeps all leaves at the same level.

As you can see here, the root must be an exception to the requirement that each node contains a minimum number of data values, since root-splitting will naturally lead to a new root node holding only one value.

B Tree Insertion Algorithm

```
InsertHelper(Val, sRoot, upVal, upChild, splitHappened) {

    NULL test, on general principles

    if at leaf {
        if NOTFULL {
            insert Val
            splitHappened = false
        }
        else {
            split off new right sibling for sRoot
            set upVal to middle value from splitting
            set upChild to new right sibling
            splitHappened = true
        }
        return
    }

    find index Idx of child to descend
    InsertHelper(Val, ptr[Idx], upVal, upChild, splitHappened)
    . . .
```

Note: this started as an implementation in C++; adapt to the language of your choice…

```
    . . .
    if ( splitHappened ) {
        if NOTFULL {
            insert upVal and upChild to sRoot
            splitHappened = false
        }
        else {
            split off new right sibling for sRoot
            set upVal to middle value from splitting
            set upChild to new right sibling
            splitHappened = true
        }
    }
    return
}
```

If we let $q = \lceil m/2 \rceil$ then we can derive an upper bound on the height of the B-tree storing n key values:

$$h \leq \log_q \left(\frac{n+1}{2} \right) + 1$$

This is very small. For example, if $m = 200$ and $n = 2{,}000{,}000$ then $h <= 4$.

But don't get too excited by this. The cost of doing a binary search of the data values in a node would be at least $\log_2(q)$, and if we do that at each level in the tree, the total cost would be

$$\Theta \left(\log(q) \cdot \log_q \left(\frac{n+1}{2} \right) \right) = \Theta \left(\log \left(\frac{n+1}{2} \right) \right) = \Theta (\log n)$$

Keep in mind that the motivation is to find a tree structure that can be efficiently stored to disk, and matching the search cost of a perfectly balanced binary tree is a plus.

It would seem that the primary concern about the cost of insertion would be the number of splits that must be performed (everything else is essentially analogous to BST insertion).

It is possible to show that as n increases, the average probability of a split is approximated by

$$\frac{1}{\lceil m/2 \rceil - 1}$$

So, for example, if m = 100 then the probability of a split is about 2%. That shouldn't be surprising.

Splitting a node is fairly expensive since about half the data values in the node must be moved to a new location, but for typical B-trees it won't be required all that often.

Deletion of a value from a node has an interesting consequence, since the number of children is related to the number of values in the node.

For a leaf node, deleting a value may drop the number of data values in the node below the mandatory floor. If that happens, the leaf must borrow a value from an adjacent sibling node if one has a value to spare, or be merged with an adjacent sibling node. But the latter will decrease the number of children the parent node has, and so a value must be moved from the parent node into the merged leaf.

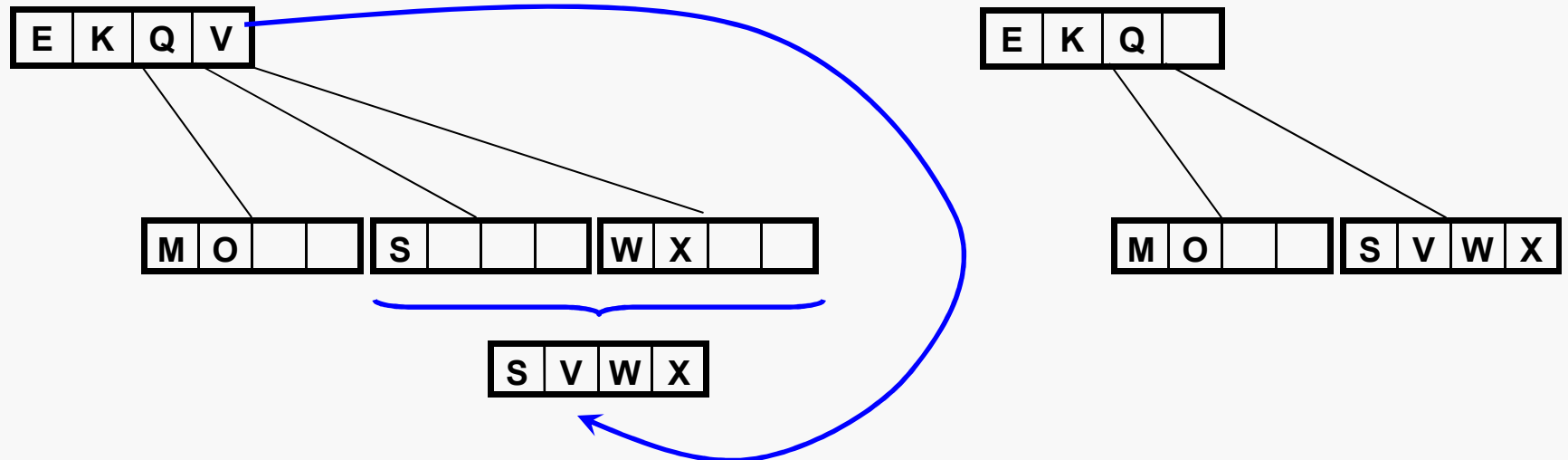Consider deleting **T** from the B-tree of order 5 below:
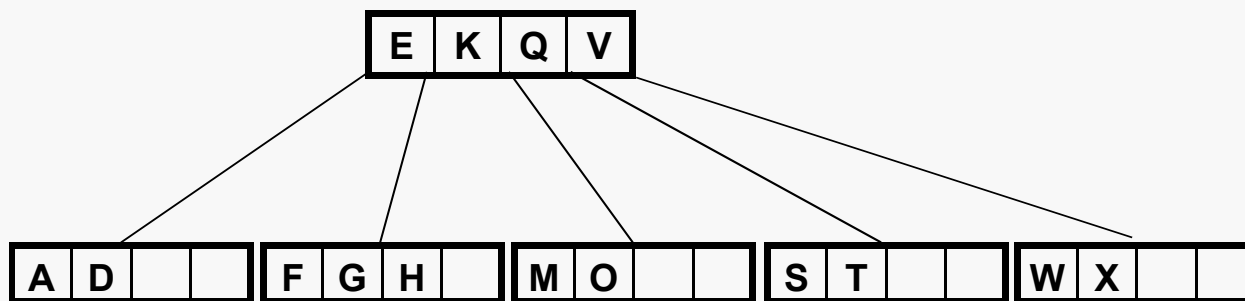
Removing **T** from the leaf causes it to "underflow".

```
              E  K  Q  V

A D        F G H      M O        S          W X
```
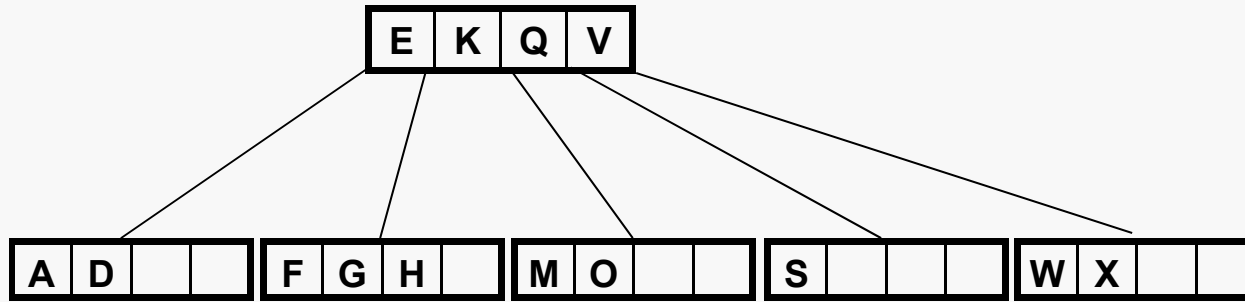
Neither sibling node has a value to spare. So we must merge with a sibling:

```
E K Q V                              E K Q

M O      S      W X                  M O       S V W X

     S V W X
```
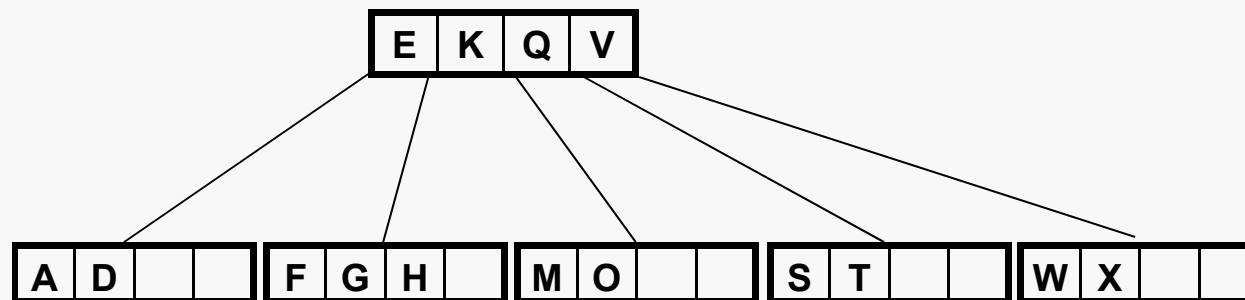
Deleting a value from an internal node is accomplished by reducing it to the former case.

Denote the value to be deleted by V_K.

The immediate predecessor of V_K, which must be in a leaf node, is borrowed to replace the value that is being deleted, and then deleted from the leaf node.
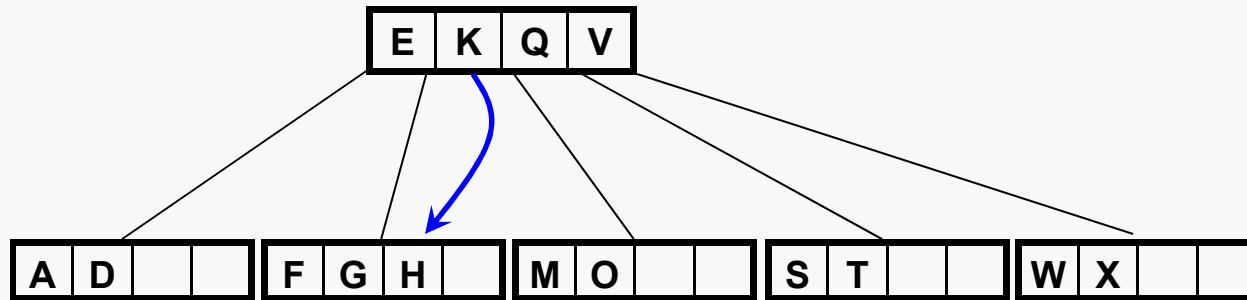
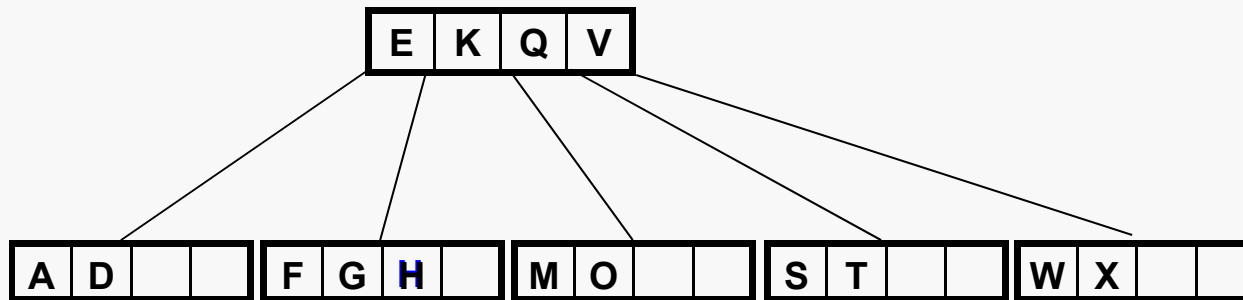Consider deleting **K** from the following B-tree of order 5:

```
                        ┌───┬───┬───┬───┐
                        │ E │ K │ Q │ V │
                        └───┴───┴───┴───┘
    ┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐  ┌───┬───┬───┬───┐
    │ A │ D │   │   │  │ F │ G │ H │   │  │ M │ O │   │   │  │ S │ T │   │   │  │ W │ X │   │   │
    └───┴───┴───┴───┘  └───┴───┴───┴───┘  └───┴───┴───┴───┘  └───┴───┴───┴───┘  └───┴───┴───┴───┘
```

The immediate predecessor of **K** is the largest value in the right-most leaf below the child that lies to the left of **K**:

The immediate predecessor is copied to replace the value being deleted and then removed from the leaf (trivial case this time):

```
DeleteHelper(Val, sRoot, underflowHappened) {

   NULL test, on general principles

   search sRoot for Val or closest predecessor
   if Val does not occur in sRoot {
      DeleteHelper(Val, appropriate child, underflowHappened)
      if success and underflowHappened {
         if can borrow {
            borrow value from appropriate child
         }
         else {
            merge appropriate children
            adjust sRoot to account for merge
            set underflowHappened
         }
      }
      return
   }
   else if sRoot is a leaf {
      delete Val from sRoot
      set underflowHappened
      return
   }
   . . .
```

Note: this started as an implementation in C++; adapt to the language of your choice…

Data Structures & Algorithms

```
   . . .
   else {
      replace Val in sRoot with closest predecessor
      DeleteHelpher(closest predecessor, left subtree from Val,
                    underflowHappened)
      if success and underflowHappened {
         if can borrow {
            borrow value from appropriate child
         }
         else {
            merge appropriate children
            adjust sRoot to account for merge
            set underflowHappened
         }
      }
      return
   }
   return
}
```

Data Structures & Algorithms

In a B tree:

- nodes are guaranteed to be (essentially) at least 50% full

- node could also be only 50% full, wasting half the data space in the nodes

- but that "wasted" space is available to service future insertions

- analysis and simulation indicates that in typical use a B tree will be about 70% full

This expectation of wasted space is a motivation for some variants of the basic B tree.

In B* trees:

- all nodes except the root are required to be at least 2/3 full rather than 1/2 full

- splitting transforms 2 nodes into 3, rather than 1 node into 2

- analysis indicates the average utilization of a B* tree will be about 81%

- can be generalized to specify a fill factor of $(n+1)/(n+2)$; a B^n tree

Knuth

In B+ trees:

- Internal nodes store only key values and pointers*.

- All records, or pointers to records, are stored in leaves.

- Commonly, the leaves are simply the logical blocks of a database file index, storing key values and offsets. In this case, many key values will occur twice in the tree, once at an internal node to guide searching, and again in a leaf.

- If the leaves are simply an index, it is common to implement the leaf level as a linked list of B tree nodes… why?

The B+ tree is the most commonly implemented variant of the B-tree family, and the structure of choice for large databases.

> *   **In small databases, it is fairly common to use a B-tree as a direct data structure, with nodes storing records.**

A sample B tree template interface is shown below:

```cpp
template <typename T> class BTreeT {
public:
   BTreeT(unsigned int Order = 3);
   ~BTreeT();

   // some public fns not shown
   bool Insert(const T& Val);
   bool Delete(const T& Val);
   void Display(ostream& Out) const;
   T* const       Find(const T& Target);
   const T* const Find(const T& Target) const;

private:
   unsigned int Order;
   BNodeT<T>*   Root;
   // some helper fns not shown
};
```

The order could be supplied as a non-type template parameter, rather than via the constructor, but you could not then use a local variable when declaring a B tree, so the approach shown here is more flexible from the client perspective.

The design of the B tree node type raises some interesting issues:

- the node must provide a way to store `Order - 1` data values

- the node must provide a way to store `Order` pointers

- the node must support binary search of the data values, so dynamically-allocated arrays are appropriate

- the node should, perhaps, provide the necessary search function, and functions to remove and add data values (and adjust the pointer array as needed)

- the node destructor must destroy the arrays

- the node should provide an `isFull()` function, and perhaps other tests such as underflow, or whether the node has "extra" values that could be "borrowed"

- the split and merge operations needed for the B tree implementation could be made the responsibility of the node

Since leaf nodes do not need pointers, there is a case for having two distinct node types. While that would save memory (or disk space), the idea will not be pursued here.

The most common approach seems to be to use one array, of dimension `Order - 1`, to store the data values, and another, of dimension `Order`, to store the pointers.

It is possible to simplify the shifting needed when inserting and removing data values by coalescing the two arrays into one, storing data/pointer pairs. For example:

```cpp
template <typename T> class Pair {
public:
    T Data;
    BNodeT<T>* Right;
    // some members perhaps not shown
};
```

The node could then store an array of `Pair` objects, of dimension `Order`, using the 0-th cell to store only a pointer (to the left-most child).

This approach also simplifies some communication issues when splitting a node.

On the other hand, this approach also adds some syntactic complexity.

Of course, the point is that we will store the nodes persistently on disk.

So, how will we lay out the node in a file?

- write alternating pointer (offset) and data values?

- write the key values as a block and the pointers as a block?

- what other values are necessary?

There are a number of options… it will be up to you to pick one.

And, of course, the nature of the data values must be taken into account.

- fixed-length simple data values?

- variable-length or otherwise complex data values

- text format or binary format?