Chapter 14

External Memory Searching

Throughout this book, we have been using the w-bit word-RAM model of computation defined in Section 1.4. An implicit assumption of this model is that our computer has a large enough random access memory to store all of the data in the data structure. In some situations, this assumption is not valid. There exist collections of data so large that no computer has enough memory to store them. In such cases, the application must resort to storing the data on some external storage medium such as a hard disk, a solid state disk, or even a network file server (which has its own external storage).

Accessing an item from external storage is extremely slow. The hard disk attached to the computer on which this book was written has an average access time of 19ms and the solid state drive attached to the computer has an average access time of 0.3ms. In contrast, the random access memory in the computer has an average access time of less than 0.000113ms. Accessing RAM is more than 2 500 times faster than accessing the solid state drive and more than 160 000 times faster than accessing the hard drive.

These speeds are fairly typical; accessing a random byte from RAM is thousands of times faster than accessing a random byte from a hard disk or solid-state drive. Access time, however, does not tell the whole story. When we access a byte from a hard disk or solid state disk, an entire *block* of the disk is read. Each of the drives attached to the computer has a block size of 4 096; each time we read one byte, the drive gives us a block containing 4 096 bytes. If we organize our data structure carefully, this

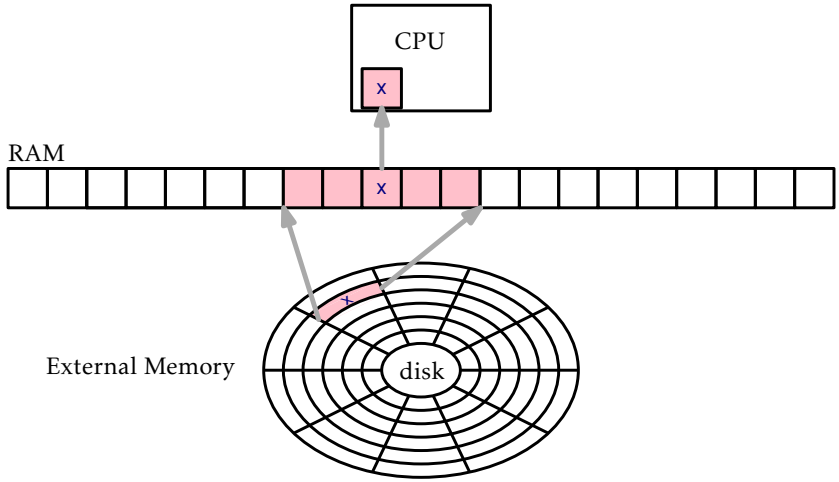

Figure 14.1: In the external memory model, accessing an individual item, x, in the external memory requires reading the entire block containing x into RAM.

means that each disk access could yield 4 096 bytes that are helpful in completing whatever operation we are doing.

This is the idea behind the *external memory model* of computation, illustrated schematically in Figure 14.1. In this model, the computer has access to a large external memory in which all of the data resides. This memory is divided into memory *blocks* each containing B words. The computer also has limited internal memory on which it can perform computations. Transferring a block between internal memory and external memory takes constant time. Computations performed within the internal memory are *free*; they take no time at all. The fact that internal memory computations are free may seem a bit strange, but it simply emphasizes the fact that external memory is so much slower than RAM.

In the full-blown external memory model, the size of the internal memory is also a parameter. However, for the data structures described in this chapter, it is sufficient to have an internal memory of size $O(B + \log_B n)$. That is, the memory needs to be capable of storing a constant number of blocks and a recursion stack of height $O(\log_B n)$. In most cases, the $O(B)$ term dominates the memory requirement. For example, even with the relatively small value $B = 32$, $B \geq \log_B n$ for all $n \leq 2^{160}$. In deci-

mal, $B \geq \log_B n$ for any

$$n \leq 1\,461\,501\,637\,330\,902\,918\,203\,684\,832\,716\,283\,019\,655\,932\,542\,976 \ .$$

14.1 The Block Store

The notion of external memory includes a large number of possible different devices, each of which has its own block size and is accessed with its own collection of system calls. To simplify the exposition of this chapter so that we can focus on the common ideas, we encapsulate external memory devices with an object called a `BlockStore`. A `BlockStore` stores a collection of memory blocks, each of size B. Each block is uniquely identified by its integer index. A `BlockStore` supports these operations:

1. `readBlock(i)`: Return the contents of the block whose index is `i`.
2. `writeBlock(i,b)`: Write contents of `b` to the block whose index is `i`.
3. `placeBlock(b)`: Return a new index and store the contents of `b` at this index.
4. `freeBlock(i)`: Free the block whose index is `i`. This indicates that the contents of this block are no longer used so the external memory allocated by this block may be reused.

The easiest way to imagine a `BlockStore` is to imagine it as storing a file on disk that is partitioned into blocks, each containing B bytes. In this way, `readBlock(i)` and `writeBlock(i,b)` simply read and write bytes $iB, \ldots, (i+1)B - 1$ of this file. In addition, a simple `BlockStore` could keep a *free list* of blocks that are available for use. Blocks freed with `freeBlock(i)` are added to the free list. In this way, `placeBlock(b)` can use a block from the free list or, if none is available, append a new block to the end of the file.

14.2 B-Trees

In this section, we discuss a generalization of binary trees, called B-trees, which is efficient in the external memory model. Alternatively, B-trees

can be viewed as the natural generalization of 2-4 trees described in Section 9.1. (A 2-4 tree is a special case of a B-tree that we get by setting $B = 2$.)

For any integer $B \geq 2$, a *B-tree* is a tree in which all of the leaves have the same depth and every non-root internal node, `u`, has at least B children and at most $2B$ children. The children of `u` are stored in an array, `u.children`. The required number of children is relaxed at the root, which can have anywhere between 2 and $2B$ children.

If the height of a B-tree is h, then it follows that the number, ℓ, of leaves in the B-tree satisfies

$$2B^{h-1} \leq \ell \leq 2(2B)^{h-1} \ .$$

Taking the logarithm of the first inequality and rearranging terms yields:

$$\begin{aligned} h &\leq \frac{\log \ell - 1}{\log B} + 1 \\ &\leq \frac{\log \ell}{\log B} + 1 \\ &= \log_B \ell + 1 \ . \end{aligned}$$

That is, the height of a B-tree is proportional to the base-B logarithm of the number of leaves.

Each node, `u`, in B-tree stores an array of keys `u.keys`$[0],\ldots,$`u.keys`$[2B-1]$. If `u` is an internal node with k children, then the number of keys stored at `u` is exactly $k-1$ and these are stored in `u.keys`$[0],\ldots,$`u.keys`$[k-2]$. The remaining $2B-k+1$ array entries in `u.keys` are set to `null`. If `u` is a non-root leaf node, then `u` contains between $B-1$ and $2B-1$ keys. The keys in a B-tree respect an order similar to the keys in a binary search tree. For any node, `u`, that stores $k-1$ keys,

$$\texttt{u.keys}[0] < \texttt{u.keys}[1] < \cdots < \texttt{u.keys}[k-2] \ .$$

If `u` is an internal node, then for every `i` $\in \{0,\ldots,k-2\}$, `u.keys[i]` is larger than every key stored in the subtree rooted at `u.children[i]` but smaller than every key stored in the subtree rooted at `u.children[i + 1]`. Informally,

$$\texttt{u.children[i]} \prec \texttt{u.keys[i]} \prec \texttt{u.children[i + 1]} \ .$$

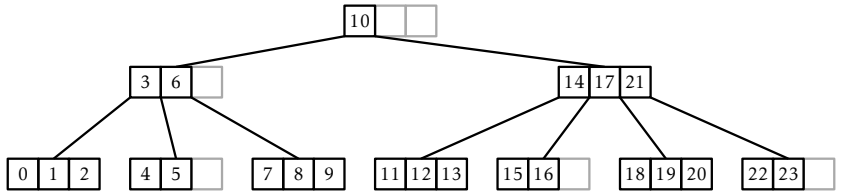

Figure 14.2: A B-tree with $B = 2$.

An example of a B-tree with $B = 2$ is shown in Figure 14.2.

Note that the data stored in a B-tree node has size $O(B)$. Therefore, in an external memory setting, the value of B in a B-tree is chosen so that a node fits into a single external memory block. In this way, the time it takes to perform a B-tree operation in the external memory model is proportional to the number of nodes that are accessed (read or written) by the operation.

For example, if the keys are 4 byte integers and the node indices are also 4 bytes, then setting $B = 256$ means that each node stores

$$(4 + 4) \times 2B = 8 \times 512 = 4096$$

bytes of data. This would be a perfect value of B for the hard disk or solid state drive discussed in the introduction to this chaper, which have a block size of 4096 bytes.

The `BTree` class, which implements a B-tree, stores a `BlockStore`, `bs`, that stores `BTree` nodes as well as the index, `ri`, of the root node. As usual, an integer, `n`, is used to keep track of the number of items in the data structure:

`BTree`

```
int n;
BlockStore<Node> bs;
int ri;
```

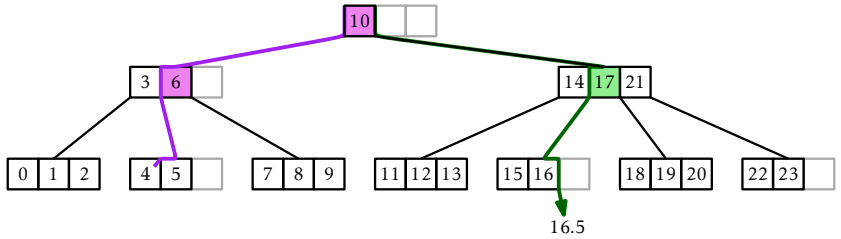

Figure 14.3: A successful search (for the value 4) and an unsuccessful search (for the value 16.5) in a *B*-tree. Shaded nodes show where the value of `z` is updated during the searches.

14.2.1 Searching

The implementation of the `find(x)` operation, which is illustrated in Figure 14.3, generalizes the `find(x)` operation in a binary search tree. The search for `x` starts at the root and uses the keys stored at a node, `u`, to determine in which of `u`'s children the search should continue.

More specifically, at a node `u`, the search checks if `x` is stored in `u.keys`. If so, `x` has been found and the search is complete. Otherwise, the search finds the smallest integer, `i`, such that `u.keys[i]` > `x` and continues the search in the subtree rooted at `u.children[i]`. If no key in `u.keys` is greater than `x`, then the search continues in `u`'s rightmost child. Just like binary search trees, the algorithm keeps track of the most recently seen key, `z`, that is larger than `x`. In case `x` is not found, `z` is returned as the smallest value that is greater or equal to `x`.

BTree

```
T find(T x) {
  T z = null;
  int ui = ri;
  while (ui >= 0) {
    Node u = bs.readBlock(ui);
    int i = findIt(u.keys, x);
    if (i < 0) return u.keys[-(i+1)]; // found it
    if (u.keys[i] != null)
      z = u.keys[i];
    ui = u.children[i];
  }
```

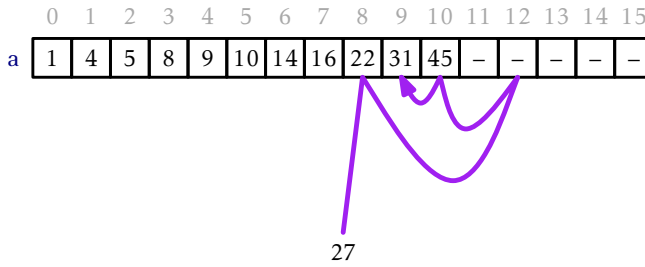

Figure 14.4: The execution of `findIt(a,27)`.

```
  return z;
}
```

Central to the `find(x)` method is the `findIt(a,x)` method that searches in a `null`-padded sorted array, `a`, for the value `x`. This method, illustrated in Figure 14.4, works for any array, `a`, where $a[0],\ldots,a[k-1]$ is a sequence of keys in sorted order and $a[k],\ldots,a[a.length-1]$ are all set to `null`. If `x` is in the array at position `i`, then `findIt(a,x)` returns $-i-1$. Otherwise, it returns the smallest index, `i`, such that $a[i] > x$ or $a[i] = null$.

BTree
```
int findIt(T[] a, T x) {
  int lo = 0, hi = a.length;
  while (hi != lo) {
    int m = (hi+lo)/2;
    int cmp = a[m] == null ? -1 : compare(x, a[m]);
    if (cmp < 0)
      hi = m;      // look in first half
    else if (cmp > 0)
      lo = m+1;    // look in second half
    else
      return -m-1; // found it
  }
  return lo;
}
```

The `findIt(a,x)` method uses a binary search that halves the search space at each step, so it runs in $O(\log(a.length))$ time. In our setting, $a.length = 2B$, so `findIt(a,x)` runs in $O(\log B)$ time.

We can analyze the running time of a B-tree `find(x)` operation both in the usual word-RAM model (where every instruction counts) and in the external memory model (where we only count the number of nodes accessed). Since each leaf in a B-tree stores at least one key and the height of a B-Tree with ℓ leaves is $O(\log_B \ell)$, the height of a B-tree that stores n keys is $O(\log_B n)$. Therefore, in the external memory model, the time taken by the `find(x)` operation is $O(\log_B n)$. To determine the running time in the word-RAM model, we have to account for the cost of calling `findIt(a,x)` for each node we access, so the running time of `find(x)` in the word-RAM model is

$$O(\log_B n) \times O(\log B) = O(\log n) \ .$$

14.2.2 Addition

One important difference between B-trees and the `BinarySearchTree` data structure from Section 6.2 is that the nodes of a B-tree do not store pointers to their parents. The reason for this will be explained shortly. The lack of parent pointers means that the `add(x)` and `remove(x)` operations on B-trees are most easily implemented using recursion.

Like all balanced search trees, some form of rebalancing is required during an `add(x)` operation. In a B-tree, this is done by *splitting* nodes. Refer to Figure 14.5 for what follows. Although splitting takes place across two levels of recursion, it is best understood as an operation that takes a node `u` containing $2B$ keys and having $2B+1$ children. It creates a new node, `w`, that adopts `u.children`$[B],\ldots,$`u.children`$[2B]$. The new node `w` also takes `u`'s B largest keys, `u.keys`$[B],\ldots,$`u.keys`$[2B-1]$. At this point, `u` has B children and B keys. The extra key, `u.keys`$[B-1]$, is passed up to the parent of `u`, which also adopts `w`.

Notice that the splitting operation modifies three nodes: `u`, `u`'s parent, and the new node, `w`. This is why it is important that the nodes of a B-tree do not maintain parent pointers. If they did, then the $B+1$ children adopted by `w` would all need to have their parent pointers modified. This would increase the number of external memory accesses from 3 to $B+4$ and would make B-trees much less efficient for large values of B.

The `add(x)` method in a B-tree is illustrated in Figure 14.6. At a high

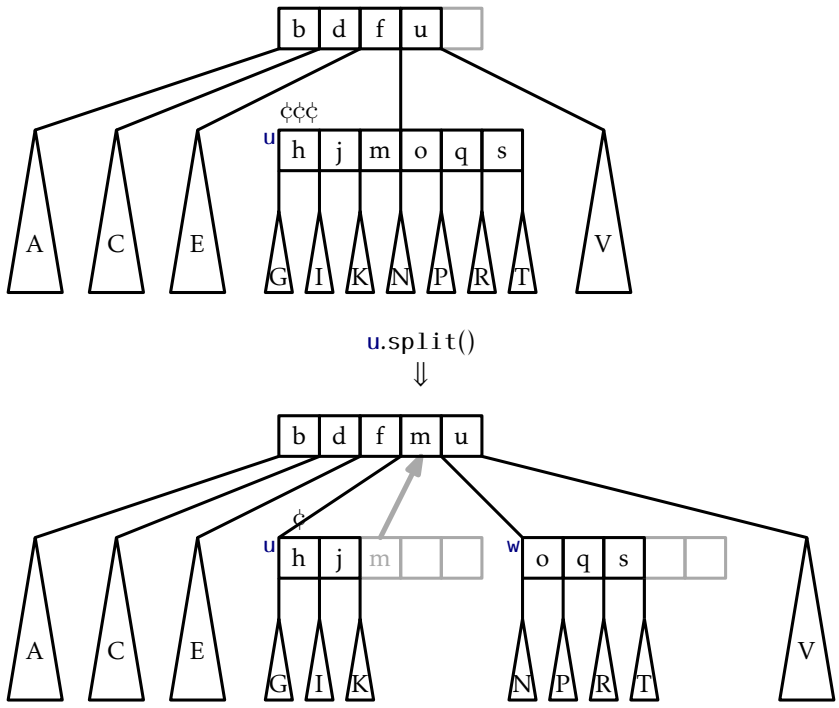

Figure 14.5: Splitting the node u in a B-tree ($B = 3$). Notice that the key u.keys[2] = m passes from u to its parent.

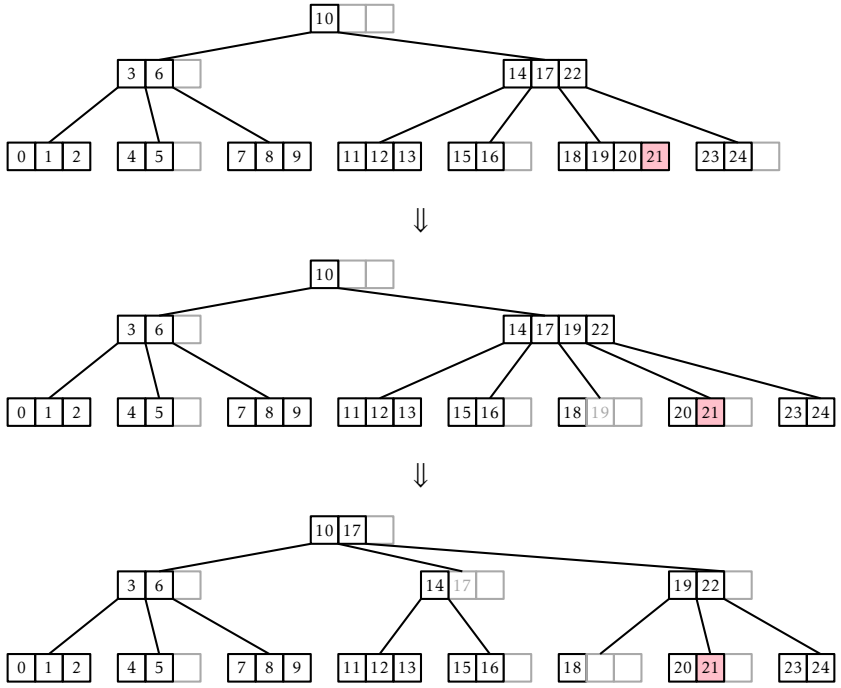

Figure 14.6: The add(x) operation in a BTree. Adding the value 21 results in two nodes being split.

level, this method finds a leaf, u, at which to add the value x. If this causes u to become overfull (because it already contained $B-1$ keys), then u is split. If this causes u's parent to become overfull, then u's parent is also split, which may cause u's grandparent to become overfull, and so on. This process continues, moving up the tree one level at a time until reaching a node that is not overfull or until the root is split. In the former case, the process stops. In the latter case, a new root is created whose two children become the nodes obtained when the original root was split.

The executive summary of the add(x) method is that it walks from the root to a leaf searching for x, adds x to this leaf, and then walks back up to the root, splitting any overfull nodes it encounters along the way. With this high level view in mind, we can now delve into the details of how this method can be implemented recursively.

The real work of `add(x)` is done by the `addRecursive(x,ui)` method, which adds the value `x` to the subtree whose root, `u`, has the identifier `ui`. If `u` is a leaf, then `x` is simply inserted into `u.keys`. Otherwise, `x` is added recursively into the appropriate child, `u'`, of `u`. The result of this recursive call is normally `null` but may also be a reference to a newly-created node, `w`, that was created because `u'` was split. In this case, `u` adopts `w` and takes its first key, completing the splitting operation on `u'`.

After the value `x` has been added (either to `u` or to a descendant of `u`), the `addRecursive(x,ui)` method checks to see if `u` is storing too many (more than $2B-1$) keys. If so, then `u` needs to be *split* with a call to the `u.split()` method. The result of calling `u.split()` is a new node that is used as the return value for `addRecursive(x,ui)`.

BTree

```
Node addRecursive(T x, int ui) {
  Node u = bs.readBlock(ui);
  int i = findIt(u.keys, x);
  if (i < 0) throw new DuplicateValueException();
  if (u.children[i] < 0) { // leaf node, just add it
    u.add(x, -1);
    bs.writeBlock(u.id, u);
  } else {
    Node w = addRecursive(x, u.children[i]);
    if (w != null) {  // child was split, w is new child
      x = w.remove(0);
      bs.writeBlock(w.id, w);
      u.add(x, w.id);
      bs.writeBlock(u.id, u);
    }
  }
  return u.isFull() ? u.split() : null;
}
```

The `addRecursive(x,ui)` method is a helper for the `add(x)` method, which calls `addRecursive(x,ri)` to insert `x` into the root of the B-tree. If `addRecursive(x,ri)` causes the root to split, then a new root is created that takes as its children both the old root and the new node created by the splitting of the old root.

BTree

```
boolean add(T x) {
  Node w;
  try {
    w = addRecursive(x, ri);
  } catch (DuplicateValueException e) {
    return false;
  }
  if (w != null) {   // root was split, make new root
    Node newroot = new Node();
    x = w.remove(0);
    bs.writeBlock(w.id, w);
    newroot.children[0] = ri;
    newroot.keys[0] = x;
    newroot.children[1] = w.id;
    ri = newroot.id;
    bs.writeBlock(ri, newroot);
  }
  n++;
  return true;
}
```

The `add(x)` method and its helper, `addRecursive(x,ui)`, can be analyzed in two phases:

Downward phase: During the downward phase of the recursion, before `x` has been added, they access a sequence of `BTree` nodes and call `findIt(a,x)` on each node. As with the `find(x)` method, this takes $O(\log_B n)$ time in the external memory model and $O(\log n)$ time in the word-RAM model.

Upward phase: During the upward phase of the recursion, after `x` has been added, these methods perform a sequence of at most $O(\log_B n)$ splits. Each split involves only three nodes, so this phase takes $O(\log_B n)$ time in the external memory model. However, each split involves moving B keys and children from one node to another, so in the word-RAM model, this takes $O(B \log n)$ time.

Recall that the value of B can be quite large, much larger than even $\log n$. Therefore, in the word-RAM model, adding a value to a B-tree can

be much slower than adding into a balanced binary search tree. Later, in Section 14.2.4, we will show that the situation is not quite so bad; the amortized number of split operations done during an add(x) operation is constant. This shows that the (amortized) running time of the add(x) operation in the word-RAM model is $O(B + \log n)$.

14.2.3 Removal

The remove(x) operation in a BTree is, again, most easily implemented as a recursive method. Although the recursive implementation of remove(x) spreads the complexity across several methods, the overall process, which is illustrated in Figure 14.7, is fairly straightforward. By shuffling keys around, removal is reduced to the problem of removing a value, x′, from some leaf, u. Removing x′ may leave u with less than $B - 1$ keys; this situation is called an *underflow*.

When an underflow occurs, u either borrows keys from, or is merged with, one of its siblings. If u is merged with a sibling, then u's parent will now have one less child and one less key, which can cause u's parent to underflow; this is again corrected by borrowing or merging, but merging may cause u's grandparent to underflow. This process works its way back up to the root until there is no more underflow or until the root has its last two children merged into a single child. When the latter case occurs, the root is removed and its lone child becomes the new root.

Next we delve into the details of how each of these steps is implemented. The first job of the remove(x) method is to find the element x that should be removed. If x is found in a leaf, then x is removed from this leaf. Otherwise, if x is found at u.keys[i] for some internal node, u, then the algorithm removes the smallest value, x′, in the subtree rooted at u.children[i + 1]. The value x′ is the smallest value stored in the BTree that is greater than x. The value of x′ is then used to replace x in u.keys[i]. This process is illustrated in Figure 14.8.

The removeRecursive(x,ui) method is a recursive implementation of the preceding algorithm:

BTree

```
boolean removeRecursive(T x, int ui) {
```

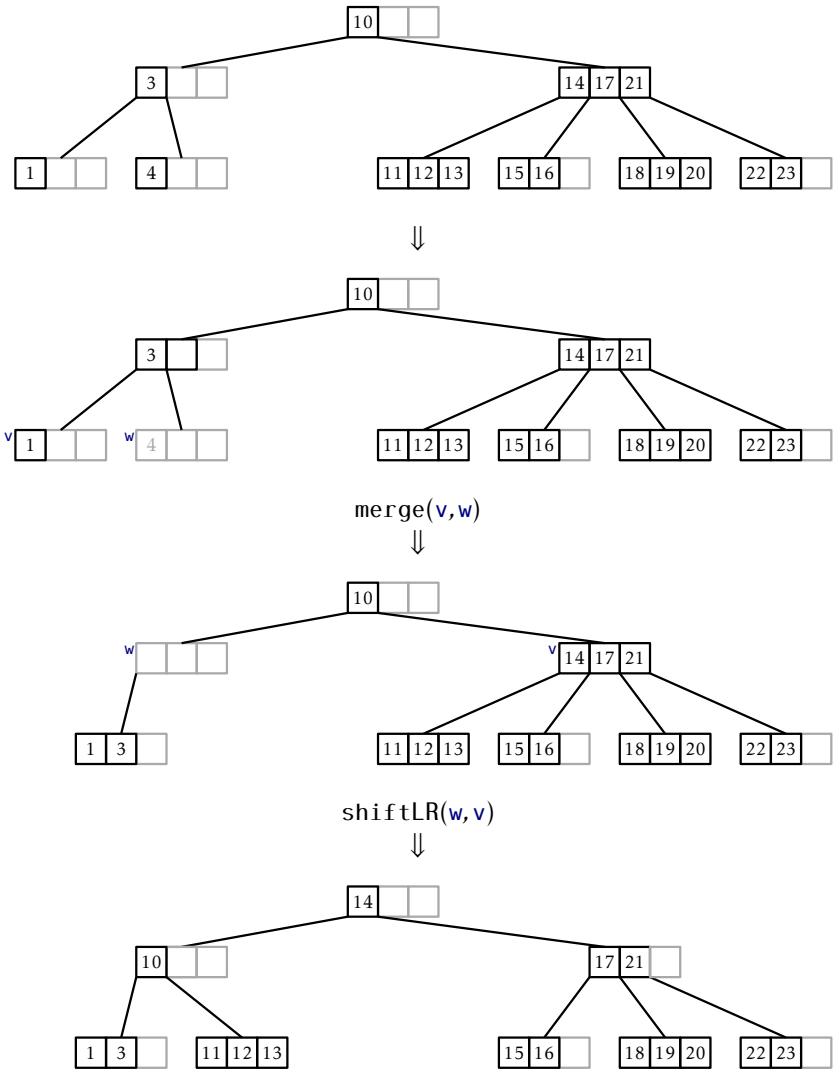

Figure 14.7: Removing the value 4 from a *B*-tree results in one merge and one borrowing operation.

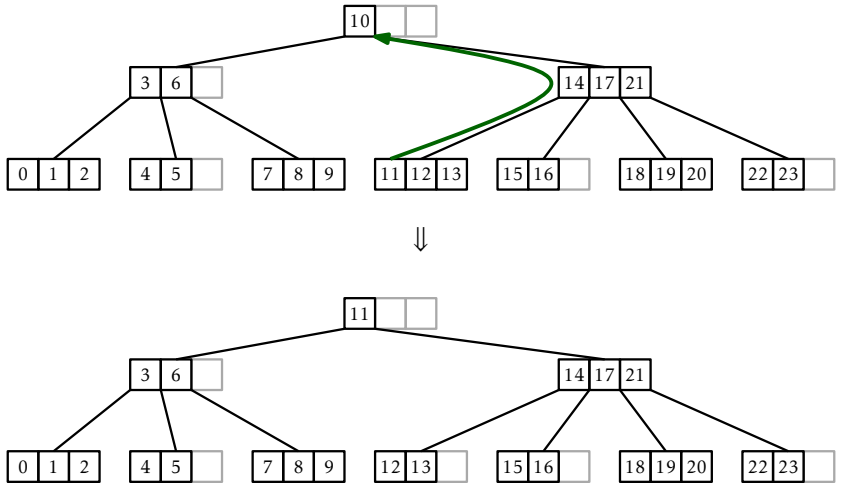

Figure 14.8: The remove(x) operation in a BTree. To remove the value x = 10 we replace it with the the value x′ = 11 and remove 11 from the leaf that contains it.

```
  if (ui < 0) return false;  // didn't find it
  Node u = bs.readBlock(ui);
  int i = findIt(u.keys, x);
  if (i < 0) { // found it
    i = -(i+1);
    if (u.isLeaf()) {
      u.remove(i);
    } else {
      u.keys[i] = removeSmallest(u.children[i+1]);
      checkUnderflow(u, i+1);
    }
    return true;
  } else if (removeRecursive(x, u.children[i])) {
    checkUnderflow(u, i);
    return true;
  }
  return false;
}
T removeSmallest(int ui) {
  Node u = bs.readBlock(ui);
  if (u.isLeaf())
```

```
    return u.remove(0);
  T y = removeSmallest(u.children[0]);
  checkUnderflow(u, 0);
  return y;
}
```

Note that, after recursively removing the value x from the ith child of u, removeRecursive(x,ui) needs to ensure that this child still has at least $B-1$ keys. In the preceding code, this is done using a method called checkUnderflow(x,i), which checks for and corrects an underflow in the ith child of u. Let w be the ith child of u. If w has only $B-2$ keys, then this needs to be fixed. The fix requires using a sibling of w. This can be either child $i+1$ of u or child $i-1$ of u. We will usually use child $i-1$ of u, which is the sibling, v, of w directly to its left. The only time this doesn't work is when $i=0$, in which case we use the sibling directly to w's right.

BTree

```
void checkUnderflow(Node u, int i) {
  if (u.children[i] < 0) return;
  if (i == 0)
    checkUnderflowZero(u, i); // use u's right sibling
  else
    checkUnderflowNonZero(u,i);
}
```

In the following, we focus on the case when $i \neq 0$ so that any underflow at the ith child of u will be corrected with the help of the $(i-1)$st child of u. The case $i=0$ is similar and the details can be found in the accompanying source code.

To fix an underflow at node w, we need to find more keys (and possibly also children), for w. There are two ways to do this:

Borrowing: If w has a sibling, v, with more than $B-1$ keys, then w can borrow some keys (and possibly also children) from v. More specifically, if v stores size(v) keys, then between them, v and w have a total of

$$B-2+\texttt{size}(\texttt{w}) \geq 2B-2$$

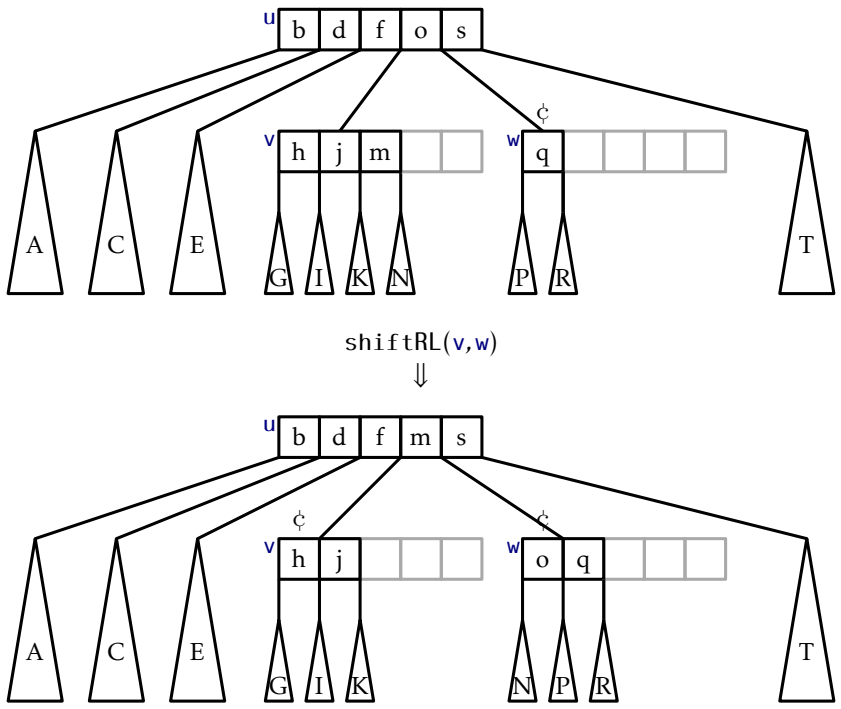

Figure 14.9: If v has more than $B-1$ keys, then w can borrow keys from v.

keys. We can therefore shift keys from v to w so that each of v and w has at least $B-1$ keys. This process is illustrated in Figure 14.9.

Merging: If v has only $B-1$ keys, we must do something more drastic, since v cannot afford to give any keys to w. Therefore, we *merge* v and w as shown in Figure 14.10. The merge operation is the opposite of the split operation. It takes two nodes that contain a total of $2B-3$ keys and merges them into a single node that contains $2B-2$ keys. (The additional key comes from the fact that, when we merge v and w, their common parent, u, now has one less child and therefore needs to give up one of its keys.)

BTree

```
void checkUnderflowNonZero(Node u, int i) {
  Node w = bs.readBlock(u.children[i]);  // w is child of u
```

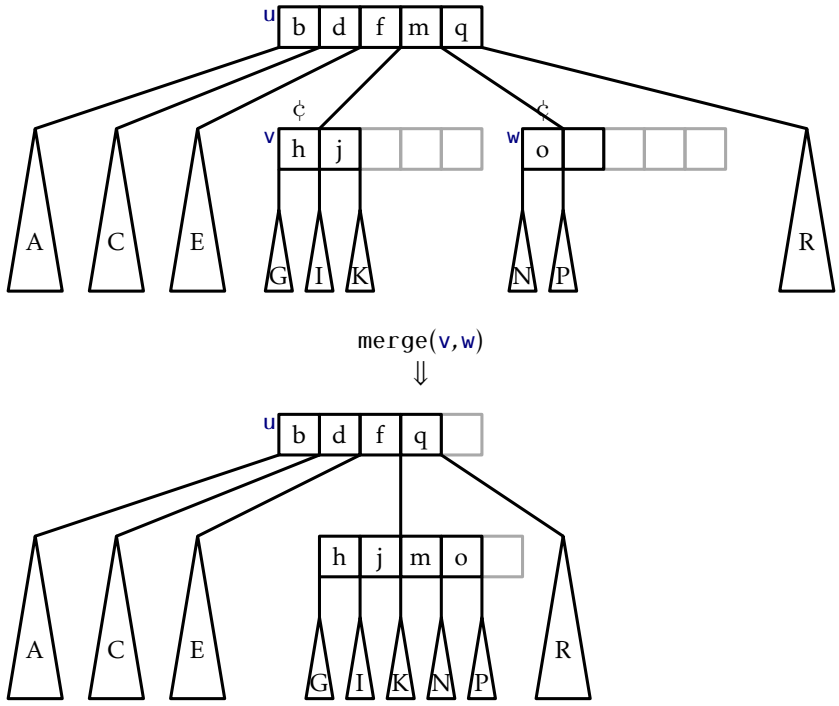

Figure 14.10: Merging two siblings v and w in a B-tree ($B = 3$).

```
  if (w.size() < B-1) {  // underflow at w
    Node v = bs.readBlock(u.children[i-1]); // v left of w
    if (v.size() > B) {  // w can borrow from v
      shiftLR(u, i-1, v, w);
    } else { // v will absorb w
      merge(u, i-1, v, w);
    }
  }
}
void checkUnderflowZero(Node u, int i) {
  Node w = bs.readBlock(u.children[i]); // w is child of u
  if (w.size() < B-1) {  // underflow at w
    Node v = bs.readBlock(u.children[i+1]); // v right of w
    if (v.size() > B) { // w can borrow from v
      shiftRL(u, i, v, w);
    } else { // w will absorb w
      merge(u, i, w, v);
      u.children[i] = w.id;
    }
  }
}
```

To summarize, the `remove(x)` method in a B-tree follows a root to leaf path, removes a key `x`' from a leaf, `u`, and then performs zero or more merge operations involving `u` and its ancestors, and performs at most one borrowing operation. Since each merge and borrow operation involves modifying only three nodes, and only $O(\log_B n)$ of these operations occur, the entire process takes $O(\log_B n)$ time in the external memory model. Again, however, each merge and borrow operation takes $O(B)$ time in the word-RAM model, so (for now) the most we can say about the running time required by `remove(x)` in the word-RAM model is that it is $O(B \log_B n)$.

14.2.4 Amortized Analysis of B-Trees

Thus far, we have shown that

1. In the external memory model, the running time of `find(x)`, `add(x)`, and `remove(x)` in a B-tree is $O(\log_B n)$.

2. In the word-RAM model, the running time of `find(x)` is $O(\log n)$ and the running time of `add(x)` and `remove(x)` is $O(B \log n)$.

The following lemma shows that, so far, we have overestimated the number of merge and split operations performed by B-trees.

Lemma 14.1. *Starting with an empty B-tree and performing any sequence of m `add(x)` and `remove(x)` operations results in at most $3m/2$ splits, merges, and borrows being performed.*

Proof. The proof of this has already been sketched in Section 9.3 for the special case in which $B = 2$. The lemma can be proven using a credit scheme, in which

1. each split, merge, or borrow operation is paid for with two credits, i.e., a credit is removed each time one of these operations occurs; and

2. at most three credits are created during any `add(x)` or `remove(x)` operation.

Since at most $3m$ credits are ever created and each split, merge, and borrow is paid for with with two credits, it follows that at most $3m/2$ splits, merges, and borrows are performed. These credits are illustrated using the ¢ symbol in Figures 14.5, 14.9, and 14.10.

To keep track of these credits the proof maintains the following *credit invariant*: Any non-root node with $B-1$ keys stores one credit and any node with $2B-1$ keys stores three credits. A node that stores at least B keys and most $2B-2$ keys need not store any credits. What remains is to show that we can maintain the credit invariant and satisfy properties 1 and 2, above, during each `add(x)` and `remove(x)` operation.

Adding: The `add(x)` method does not perform any merges or borrows, so we need only consider split operations that occur as a result of calls to `add(x)`.

Each split operation occurs because a key is added to a node, `u`, that already contains $2B-1$ keys. When this happens, `u` is split into two nodes, `u'` and `u''` having $B-1$ and B keys, respectively. Prior to this operation, `u` was storing $2B-1$ keys, and hence three credits. Two of these credits can

be used to pay for the split and the other credit can be given to u′ (which has $B-1$ keys) to maintain the credit invariant. Therefore, we can pay for the split and maintain the credit invariant during any split.

The only other modification to nodes that occur during an add(x) operation happens after all splits, if any, are complete. This modification involves adding a new key to some node u′. If, prior to this, u′ had $2B-2$ children, then it now has $2B-1$ children and must therefore receive three credits. These are the only credits given out by the add(x) method.

Removing: During a call to remove(x), zero or more merges occur and are possibly followed by a single borrow. Each merge occurs because two nodes, v and w, each of which had exactly $B-1$ keys prior to calling remove(x) were merged into a single node with exactly $2B-2$ keys. Each such merge therefore frees up two credits that can be used to pay for the merge.

After any merges are performed, at most one borrow operation occurs, after which no further merges or borrows occur. This borrow operation only occurs if we remove a key from a leaf, v, that has $B-1$ keys. The node v therefore has one credit, and this credit goes towards the cost of the borrow. This single credit is not enough to pay for the borrow, so we create one credit to complete the payment.

At this point, we have created one credit and we still need to show that the credit invariant can be maintained. In the worst case, v's sibling, w, has exactly B keys before the borrow so that, afterwards, both v and w have $B-1$ keys. This means that v and w each should be storing a credit when the operation is complete. Therefore, in this case, we create an additional two credits to give to v and w. Since a borrow happens at most once during a remove(x) operation, this means that we create at most three credits, as required.

If the remove(x) operation does not include a borrow operation, this is because it finishes by removing a key from some node that, prior to the operation, had B or more keys. In the worst case, this node had exactly B keys, so that it now has $B-1$ keys and must be given one credit, which we create.

In either case—whether the removal finishes with a borrow operation or not—at most three credits need to be created during a call to remove(x)

to maintain the credit invariant and pay for all borrows and merges that occur. This completes the proof of the lemma. □

The purpose of Lemma 14.1 is to show that, in the word-RAM model the cost of splits, merges and joins during a sequence of m `add(x)` and `remove(x)` operations is only $O(Bm)$. That is, the amortized cost per operation is only $O(B)$, so the amortized cost of `add(x)` and `remove(x)` in the word-RAM model is $O(B + \log n)$. This is summarized by the following pair of theorems:

Theorem 14.1 (External Memory B-Trees)**.** *A `BTree` implements the `SSet` interface. In the external memory model, a `BTree` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log_B n)$ *time per operation.*

Theorem 14.2 (Word-RAM B-Trees)**.** *A `BTree` implements the `SSet` interface. In the word-RAM model, and ignoring the cost of splits, merges, and borrows, a `BTree` supports the operations* `add(x)`, `remove(x)`, *and* `find(x)` *in* $O(\log n)$ *time per operation. Furthermore, beginning with an empty `BTree`, any sequence of m* `add(x)` *and* `remove(x)` *operations results in a total of* $O(Bm)$ *time spent performing splits, merges, and borrows.*

14.3 Discussion and Exercises

The external memory model of computation was introduced by Aggarwal and Vitter [4]. It is sometimes also called the *I/O model* or the *disk access model*.

B-Trees are to external memory searching what binary search trees are to internal memory searching. B-trees were introduced by Bayer and McCreight [9] in 1970 and, less than ten years later, the title of Comer's ACM Computing Surveys article referred to them as ubiquitous [15].

Like binary search trees, there are many variants of B-Trees, including B^+-trees, B^*-trees, and counted B-trees. B-trees are indeed ubiquitous and are the primary data structure in many file systems, including Apple's HFS+, Microsoft's NTFS, and Linux's Ext4; every major database system; and key-value stores used in cloud computing. Graefe's recent survey [36] provides a 200+ page overview of the many modern applications, variants, and optimizations of B-trees.

B-trees implement the SSet interface. If only the USet interface is needed, then external memory hashing could be used as an alternative to *B*-trees. External memory hashing schemes do exist; see, for example, Jensen and Pagh [43]. These schemes implement the USet operations in $O(1)$ expected time in the external memory model. However, for a variety of reasons, many applications still use *B*-trees even though they only require USet operations.

One reason *B*-trees are such a popular choice is that they often perform better than their $O(\log_B n)$ running time bounds suggest. The reason for this is that, in external memory settings, the value of *B* is typically quite large—in the hundreds or even thousands. This means that 99% or even 99.9% of the data in a *B*-tree is stored in the leaves. In a database system with a large memory, it may be possible to cache all the internal nodes of a *B*-tree in RAM, since they only represent 1% or 0.1% of the total data set. When this happens, this means that a search in a *B*-tree involves a very fast search in RAM, through the internal nodes, followed by a single external memory access to retrieve a leaf.

Exercise 14.1. Show what happens when the keys 1.5 and then 7.5 are added to the *B*-tree in Figure 14.2.

Exercise 14.2. Show what happens when the keys 3 and then 4 are removed from the *B*-tree in Figure 14.2.

Exercise 14.3. What is the maximum number of internal nodes in a *B*-tree that stores n keys (as a function of n and *B*)?

Exercise 14.4. The introduction to this chapter claims that *B*-trees only need an internal memory of size $O(B + \log_B n)$. However, the implementation given here actually requires more memory.

1. Show that the implementation of the add(x) and remove(x) methods given in this chapter use an internal memory proportional to $B \log_B n$.

2. Describe how these methods could be modified in order to reduce their memory consumption to $O(B + \log_B n)$.

Exercise 14.5. Draw the credits used in the proof of Lemma 14.1 on the trees in Figures 14.6 and 14.7. Verify that (with three additional credits)

it is possible to pay for the splits, merges, and borrows and maintain the credit invariant.

Exercise 14.6. Design a modified version of a B-tree in which nodes can have anywhere from B up to $3B$ children (and hence $B-1$ up to $3B-1$ keys). Show that this new version of B-trees performs only $O(m/B)$ splits, merges, and borrows during a sequence of m operations. (Hint: For this to work, you will have to be more agressive with merging, sometimes merging two nodes before it is strictly necessary.)

Exercise 14.7. In this exercise, you will design a modified method of splitting and merging in B-trees that asymptotically reduces the number of splits, borrows and merges by considering up to three nodes at a time.

1. Let u be an overfull node and let v be a sibling immediately to the right of u. There are two ways to fix the overflow at u:

   (a) u can give some of its keys to v; or

   (b) u can be split and the keys of u and v can be evenly distributed among u, v, and the newly created node, w.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

2. Let u be an underfull node and let v and w be siblings of u There are two ways to fix the underflow at u:

   (a) keys can be redistributed among u, v, and w; or

   (b) u, v, and w can be merged into two nodes and the keys of u, v, and w can be redistributed amongst these nodes.

   Show that this can always be done in such a way that, after the operation, each of the (at most 3) affected nodes has at least $B+\alpha B$ keys and at most $2B-\alpha B$ keys, for some constant $\alpha > 0$.

3. Show that, with these modifications, the number of merges, borrows, and splits that occur during m operations is $O(m/B)$.

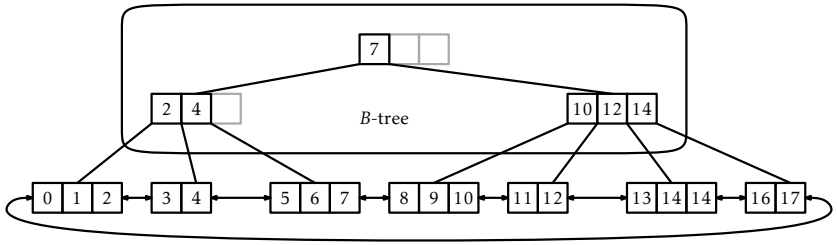

Figure 14.11: A B^+-tree is a B-tree on top of a doubly-linked list of blocks.

Exercise 14.8. A B^+-tree, illustrated in Figure 14.11 stores every key in a leaf and keeps its leaves stored as a doubly-linked list. As usual, each leaf stores between $B-1$ and $2B-1$ keys. Above this list is a standard B-tree that stores the largest value from each leaf but the last.

1. Describe fast implementations of `add(x)`, `remove(x)`, and `find(x)` in a B^+-tree.

2. Explain how to efficiently implement the `findRange(x,y)` method, that reports all values greater than `x` and less than or equal to `y`, in a B^+-tree.

3. Implement a class, `BPlusTree`, that implements `find(x)`, `add(x)`, `remove(x)`, and `findRange(x,y)`.

4. B^+-trees duplicate some of the keys because they are stored both in the B-tree and in the list. Explain why this duplication does not add up to much for large values of B.