Algorithms for Memory Hierarchies
Lecture 3

Lecturer: Nodari Sitchinava
Scribes: Mateus Grellert, Robin Rehrmann

- Last time: B-trees
- Today: Persistent B-trees

1 Persistent B-trees

When it comes to (a,b)-trees, in general, and B-trees - which are (a,b)-trees with $a = \frac{B}{4}$ and $b = B$ (Figure 1) - specifically, one could wonder, what these trees are used for.

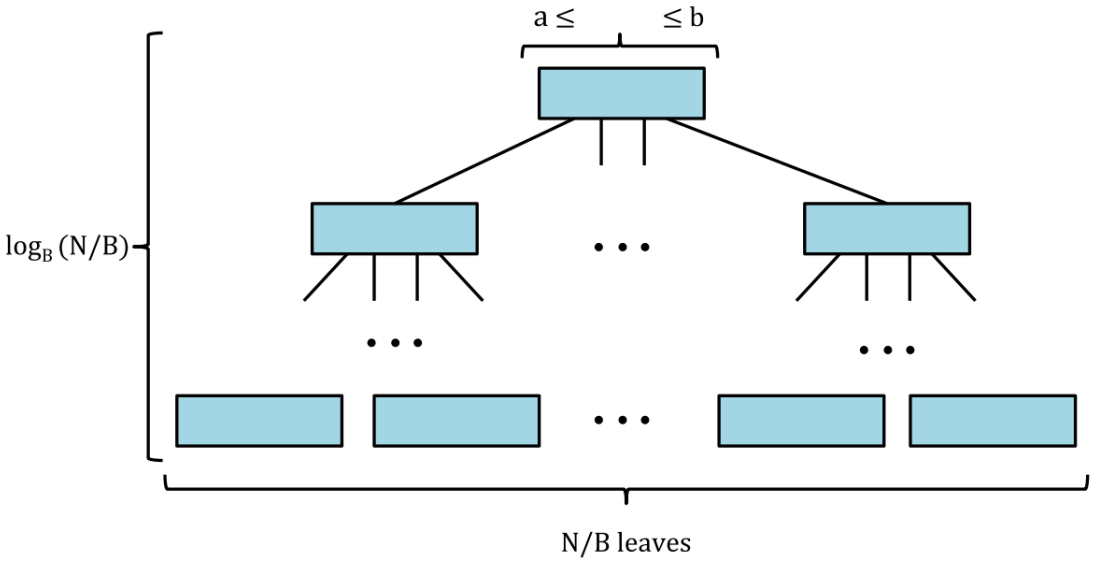

Figure 1: A B-tree.

One of the fields in which B-trees are commonly used are databases. Databases use B-trees to index data, because they provide an efficient way, to insert, search and update data.

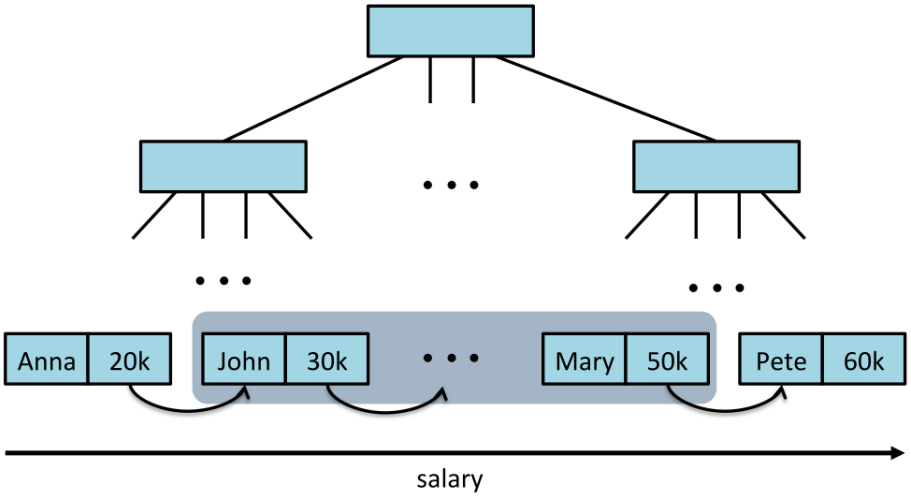

Figure 2: An example of range search using a B-tree. When searching for employees with salaries in the range between x and y, search for employee with salary x in the B-tree and then move onward on the data until reaching employee having salary y (assuming the nodes in the B-tree are stored according to the employee salaries). The I/O complexity for this search is then $O(\log_B N + \frac{K}{B})$.

Databases are used in all kinds of manner. For example, one could ask about the salary of different people over a timespan (See Figure 2). But what if a person has left the company during that timespan? If such database keeps only the current state, the mentioned person would not be found. Therefore, we would like to keep the database changes over time. How do we achieve this? There are two different solutions:

- Keep copies of different versions of databases.

  This solution copies the database after a fixed number of changes or a defined timespan. When looking for results over time, one would need to travel through a tree of versions, to find the copy of the database that pertains to the targeted time. The problem here is: when are the databases supposed to be copied? It is very expensive to copy the whole database after every change is made, but, on the other hand, it might not be good enough to copy it every thousand updates. Besides that, many identical entries would exist within that database. Thus, after N updates to a database of size $O(N)$ the space complexity for managing all copies can be as large as $O(N^2)$.

- Keep just the changes.

  The other solution is to just store the nodes that have been updated separately. Since this solution is much more space-efficient, it will be now explained in more detail.

1.1 Types of persistent data structure

There are two different types of persistent data structures, which differ from each other when it comes to the possibility of updating and querying past versions of the data structure.

Fully persistent data structures allow you to query and update any version in the past. This is a quite complicated algorithm and most of the time not needed. What is more important is the second type of persistent data structure.

Partially persistent data structures also allow you to query any version in the past, but only the most recent version can be updated. Since this is a simpler algorithm, and since it also provides anything one mostly needs, today we will talk about partially persistent B-trees.

1.2 Partially persistent B-trees

In order to control the different versions, each insertion and deletion operation on the tree is augmented with a timestamp of when it is performed, like shown on the table below.

	`insert(x)`	`insert(y)`	`delete(y)`	`insert(z)`	`delete(x)`
t	1	2	3	4	5

A naive implementation of a partially persistent B-tree consists of creating copies of the B-tree prior to each update, as well as a version access data structure – which is also a B-tree – to discover the right version for each query. This is illustrated in Figure 3.

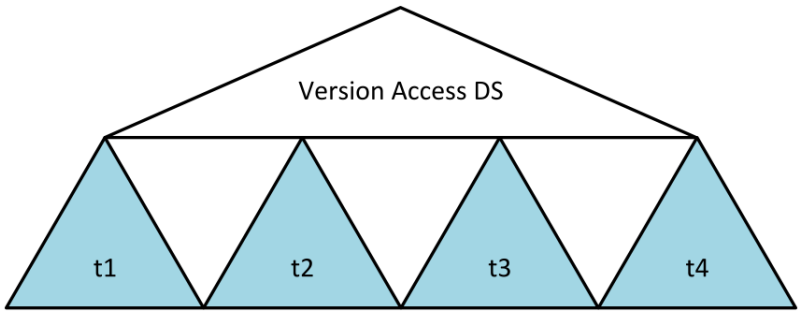

Figure 3: What the tree looked like, if the tree was copied each time something changed.

However, this approach would require too much space, since much data is replicated, so a more sophisticated approach is required. Instead of making several copies of a tree

whenever an update occurs, only new information should alter the structure. In order to do so, each node must contain its existence interval $[t_i, t_j]$. E.g. the element `x` in the table above was stored as `x[1, 5]`. The resulting structure is not a tree, but a Directed Acyclic Graph (DAG), topped by a version control structure, which is, again, also a B-tree, as depicted in Figure 4.

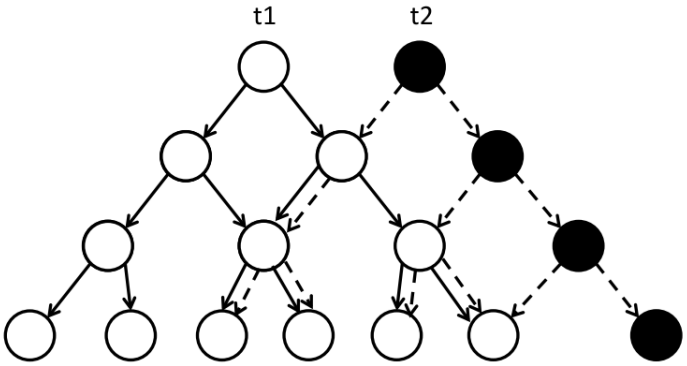

Figure 4: The Direct Acyclic Graph (DAG) returning the root node of the selected timestamp of the B-tree.

At any given time, the nodes that are alive constitute a valid B-tree, i.e., each node contains $[\frac{B}{4}, B]$ living nodes. Each new node must follow the new node invariant, which states that a new node contains only elements that are alive.

1.2.1 The new node invariant

Now we introduce an invariant for new nodes. As previously mentioned, new nodes always contain only alive elements. Additionally, whenever we create a new node we restrict the number of elements n in the new node to be $\frac{3}{8}B \leq |n| \leq \frac{7}{8}B$. In other words, the number of elements is at least $\frac{3}{8}B$, instead of $\frac{1}{4}B$, and at most $\frac{7}{8}B$, instead of B.

The new node invariants guarantee linear space of the persistent B-tree. But before we analyze the space and I/O complexity, let us first describe the operations on the persistent B-tree.

1.3 Update operations in the Persistent B-trees

We will now describe how to perform insertion and deletion in the persistent B-tree.

1.3.1 Inserting an element into the persistent B-tree

When inserting an element into the B-tree, we first find the leaf in which the element should be placed at. Finding this leaf is done with the `search` method (see Program 1).

To do this, we first query the version access structure to the root that is valid at time t. Afterwards, within the returned tree we search for the leaf that will hold the element to be inserted. Since we search in B-trees, the cost for this operation so far is $2 \cdot O(\log_B N) = O(\log_B N)$.

```
1  search(x,t)
2      vas   = get_version_access_structure();
3      btree = vas.find_root_at_time(t);
4      return btree.find(x);
5
6  insert(x,t)
7      l = search(x,t);
8      l.insert(x);
9
10     if( |l| > B )
11         l' = copy_alive_elements(l); // 1/4 B ≤ l' ≤ B+1
12         set_timestamp(l',t);
13         mark_as_deleted(l);
14         update(parent(l), l,l'); // update pointers of parent node
15         rebalance(l');
16
17 rebalance(v) // 1/4 B ≤ v ≤ B+1
18     if( 3/8 B ≤ |v| ≤ 7/8 B )
19             return;
20     if( |v| > 7/8 B )
21             strong_overflow(v);
22     if( |v| < 3/8 B )
23             strong_underflow(v);
24
25     rebalance(parent(v));
```

Program 1: Implementations of the methods `search` and `insert` and helper method `rebalance`

After inserting element x into leaf l, that leaf may need to be rebalanced. This is done in Program 2. Therefore l is marked as deleted inside of its parent and all its alive elements are copied into a new leaf l'. This new leaf is sent to method `rebalance`, in which we first need to check whether the size of the new node (i.e. its number of elements) is between $\frac{3}{8}B$ and $\frac{7}{8}B$. If so, nothing happens.

Otherwise, if $|v'| > \frac{7}{8}B$, the method `strong_overflow` is called, which splits the given node into two nodes of the same size (± 1, depending on, whether v is even, or not) and updates the parent of v.

Instead, if $|v'| < \frac{3}{8}B$, the method `strong_underflow` is called. This fuses the alive elements of the sibling v' of v and v into a single node v. Since that sibling had at least $\frac{1}{4}B$ alive elements and the new node has $\frac{1}{4}B - 1$ alive elements, the fused node now has at least $\frac{1}{2}B - 1$ elements and our invariant of having at least $\frac{3}{8}B$ elements is satisfied. If

the new node contains less than $\frac{7}{8}B$ elements, all that remains to do is update pointers in the parent node and rebalance the parent node if needed. On the other hand, because the sibling may hold B elements, the new node might contain too many elements. If so, it again needs to be split up into two new nodes having equal number of elements (± 1 depending on the parity of B).

```
1  /* This method is called, if |v| > (7/8)B. */
2  strong_overflow(v)
3      create v', v'' so that |v'| = ⌊|v|/2⌋, |v''| = ⌈|v|/2⌉ ≥ (3/8)B
4      update(parent(v), v, v',v''); // update pointers of parent node
5
6      /* parent(v)'s size increased by 1, so recursively rebalance */
7      rebalance(parent(v));
8
9  /* This method is called, if (1/4)B ≤ |v| < (3/8)B. */
10 strong_underflow(v)
11     v' = get_sibling(v);
12
13     /* Create a copy of sibling without dead elements. */
14     v'' = copy_alive_elements(v'); // |v''| ≥ (1/4)B
15     mark_as_dead(v');
16
17     /* Fuse v and v'' */
18     v = fuse(v, v'') // |v| ≥ (1/2)B − 1
19
20     if( |v| < (7/8)B )
21         update(parent(v), v,v'); // update pointers of parent node
22
23         /* parent(v)'s size decreased by 1, so recursively rebalance */
24         rebalance(parent(v));
25     else
26         // |u|,|u'| ≥ (3/8)B
27         u,u' = split(v);
28         update(parent(v), v,u,u'); // update pointers of parent node
```

Program 2: Implementations of the helper methods `strong_overflow` and `strong_underflow`.

1.3.2 Method `delete` for removing an element from the persistent B-tree

When deleting an element from the tree, we first search for the leaf holding that element (Program 3). Within that returned leaf the element to be deleted is marked as deleted, because, since we want to keep the old versions, we do not want to erase it completely.

Afterwards, it needs to be checked whether that node contains a valid number of alive elements, i.e., the number of alive (meaning the elements that have not been marked as deleted) must be at least $\frac{1}{4}B$. If not so, a copy of l is created, that contains all the alive

```
1  delete(x,t)
2  {
3          l = search(x);
4          // do not delete x within l, just mark it as deleted.
5          l.mark_as_deleted(x);
6
7          // now check for the number of active elements
8          if( |alive_elements(l)| ≤ 1/4 B )
9                  // Create a copy of l. The copy does not
10                 // hold dead elements.
11                 l' = copy_alive_elements(l);
12
13                 // because |l'| < 3/8 B
14                 strong_underflow(l');
15
16         // mark as deleted in parent
17         update(parent(l));
18         rebalance(parent(l), l,l');
19 }
```

Program 3: Implementations of `delete`. This function also uses helper functions, implemented in Program 2.

elements of l only. With this copy l', we call the method `strong_underflow`, because l' cannot hold $\frac{3}{8}B$ elements, since the original l contained less than $\frac{1}{4}B$ alive elements.

Afterwards, l needs to be marked as deleted inside of its parent, and the parent is rebalanced, if needed.

1.4 Analysis

Let us analyze the I/O-complexity of N operations on a persistent B-tree.

Search When searching an element after N operations, the I/O-complexity of that search is $O(\log_B N)$ I/Os. That is, because the search within the *Version Access Data structure* is $O(\log_B N)$ I/Os and the search within the returned B-tree is also $O(\log_B N)$ I/Os, which leads to a total complexity of $O(\log_B N)$ I/Os.

Insert/delete When inserting or deleting an element into or out of the tree, we first have to search this element (we now know that the complexity of the search is $O(\log_B N)$ I/Os). When the node is found, the insert/delete operation takes $O(1)$ I/Os in that node (since rebalancing touches at most two nodes of one block each per level). In the worst case, every predecessor of such node, up to the root, need to be updated with the same number of constant I/Os, so, in the worst case, this takes $O(1) \cdot O(\log_B N)$. So all in all

the complexity of inserting or deleting an element is $O(\log_B N) + O(1) \cdot O(\log_B N) = O(\log_B N)$.

Now let us have a look at the space required by the data structure. We will show that the space complexity for this scenario is $O(\frac{N}{B})$ blocks. And this is where our new node invariants come in.

When doing an update, we do not free any memory. So how do we guarantee that eventually after many copies are performed the data structure does not take up too much space?

When a new node is created as a copy of an old node, it has at least $\frac{3}{8}B$ elements (by the new node invariant), so it takes at least $\frac{1}{8}B$ deletes from this node to create a new copy from it again. On the other hand, a new node has at most $\frac{7}{8}B$ elements, so it takes at least $\frac{1}{8}B$ updates, to create a new copy from it, again. So all in all, it takes at least $\frac{1}{8}B$ operations on a new node, to create a new copy and "waste" space, again.

So, after N operations, we have created less than $\frac{N}{\frac{1}{8}B}$ copies of that node. Since a node is only copied, when its children are updated, the parent of this node is copied less than $\frac{N}{(\frac{1}{8}B)^2}$ times after N operations. Finally, when looking at the total amount of space on all levels of the tree, the space complexity can be represented by the following equation:

$$\sum_{i=1}^{\log_B N} \frac{N}{(\frac{1}{8}B)^i} = O\left(\frac{N}{B}\right) blocks$$

Therefore, the new node invariant is introduced in order to guarantee that not too much space is used, since linear space complexity is achieved by using this rule.

1.5 Persistent B-trees Applications

The problem of finding intersections in a set of orthogonal line segments can be modeled with a Persistent B-tree. Each begin/end coordinate can be considered the same as the timestamps t_i and t_j. Figure 5 is shown, to illustrate this.

Here, we query via SQL in past versions (as we have already seen). Having B-trees, the I/O complexity for this query is $O(\log_B N + \frac{T}{B})$. So for N horizontal segments the total complexity is $O(N \log_B N + \frac{T}{B})$ I/Os.

In two lectures, though, we will see, how to get from $O(N \log_B N + \frac{T}{B})$ I/Os to $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{T}{B})$ I/Os, which is a lot fewer I/Os, since $N \gg sort(N) = O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$.

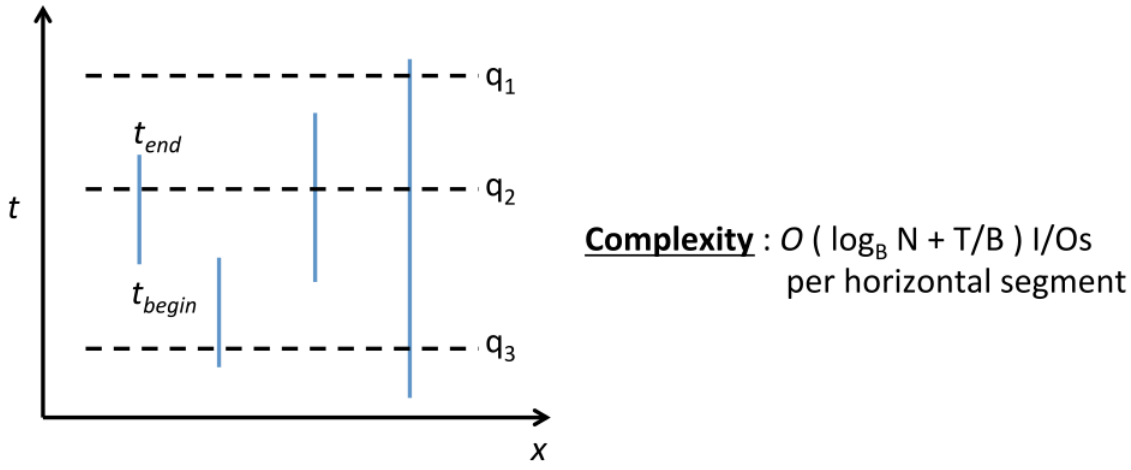
t
x
t_{end}
t_{begin}
q_1
q_2
q_3
Complexity : O ($\log_B$ N + T/B) I/Os
per horizontal segment

Figure 5