Lecture 17: B-Trees

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Introduction

Balanced BSTs such as RBTs are great for data structures that can fit into the main memory of the computer. But what happens when we need to use external storage?

Here are some approximate speeds and sizes for current technologies:

Type	Data Speed	Size	Min addressable unit
DDR RAM	12.8 GB/s	<128 GB	8 bytes
SSD	500 MB/s	< 2 TB	256 KB - 4 MB
Hard Drive	100 MB/s	< 10 TB	4 KB

For storage that has relatively slow read/write speeds, it makes sense to design data structures that minimise the number of reads/writes in order to access the data.

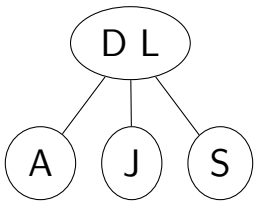
It also makes sense to have data structures that use the minimum addressable unit as their base node size.

B-trees do both and are commonly used for database applications and for file systems.

B-trees

A B-tree is an extension of a BST - instead of up to 2 children, a B-tree can have up to m children for some pre-specified integer m (called the order of the B-tree). m is typically chosen so that a B-tree node will take up one page on the drive.

For example:



Definition

A B-tree of minimum degree t satisfies the following properties:

1. Every node has at at least $t - 1$ and at most $2t - 1$ keys, except for the root.
2. Every non-leaf node (except root) has at least t and at most $2t$ children ($m = 2t$).
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k - 1$ keys.
5. All leaves appear at the same level.

Searching

Searching in a B-tree is straight-forward:

```
1: function B-Tree-Search(Node x, Key k)
2:     i ← 0
3:     while i < numkeys(x) and k > x.keys[i] do
4:         i ← i + 1
5:     end while
6:     if i < numkeys(x) and k = x.keys[i] then return (x, i)
7:     end if
8:     if leaf(x) then
9:         return (x, NULL)
10:    else
11:        return B-Tree-Search(x.child[i], k)
12:    end if
13: end function
```

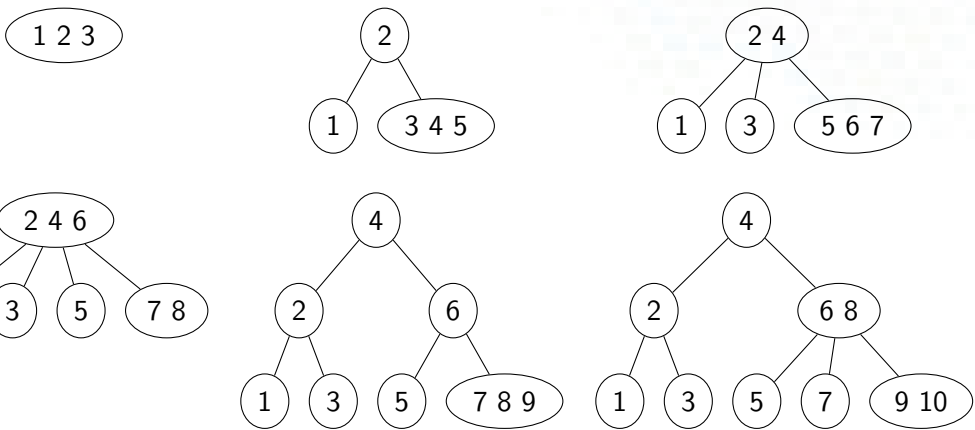

Inserting

```
1:  procedure B-Tree-Insert(Node x, Key k)
2:      find i such that x.keys[i] > k or i >=numkeys(x)
3:      if x is a leaf then
4:          Insert k into x.keys at i
5:      else
6:          if x.child[i] is full then
7:              Split x.child[i]
8:              if k > x.key[i] then
9:                  i ← i + 1
10:             end if
11:         end if
12:         B-Tree-Insert(x.child[i], k)
13:     end if
14: end procedure
```

Note: special case for splitting the root omitted for brevity.

Insert 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

For $t = 2$ the maximum number of keys in a node is 3 and the maximum number of children is 4.



Deleting

Deletion from a B-tree is a bit more complicated than insertion because a key may be deleted from any node, not just a leaf. Deletion from an internal node requires that the node's children be rearranged.

Just as we had to ensure that nodes didn't get too big due to insertion, we have to ensure that nodes don't get too small (fewer than $t - 1$ keys) due to deletion.

This is done by ensuring that, before a key is deleted from a node, that node has at least t keys —— which may mean that we have to move an extra key into a node before we can delete anything from the node. There are two ways of moving in an extra node —— we may borrow a key from a nearby node that has more than it needs, or if we can't borrow then we may merge two nodes that have no keys to spare.

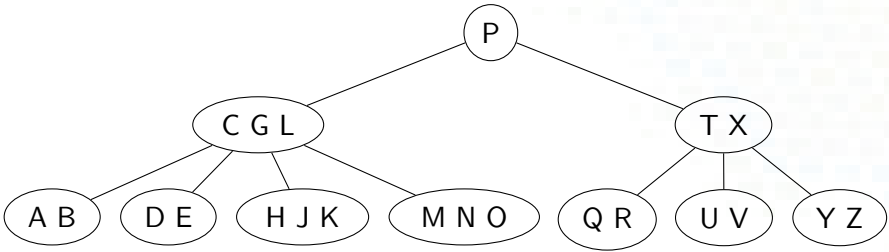
To delete key k, we search from the root for the node containing k, and strengthen each node we visit on the way if it has fewer than t keys.

Deleting

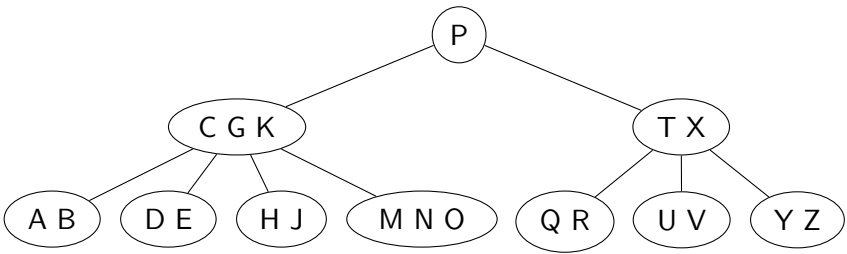
There are 3 cases for deleting from a B-tree. We reach these cases via recursion. As we recurse down the tree, we are checking which of the conditions we are in and recursively calling delete as necessary. Assume we have reached node x:

1. x is a leaf and contains the key (it will have at least t keys). This is trivial - just delete the key.
2. x is an internal node and contains the key. There are 3 subcases:
   - 2.1 predecessor child node has at least t keys
   - 2.2 successor child node has at least t keys
   - 2.3 neither predecessor nor successor child has t keys
3. x is an internal node, but doesn't contain the key. Find the child subtree of x that contains the key if it exists (call the child c). There are three subcases:
   - 3.1 c has at least t keys. Simply recurse to c.
   - 3.2 c has $t-1$ keys and one of its siblings has t keys.
   - 3.3 c and both siblings have $t-1$ keys.

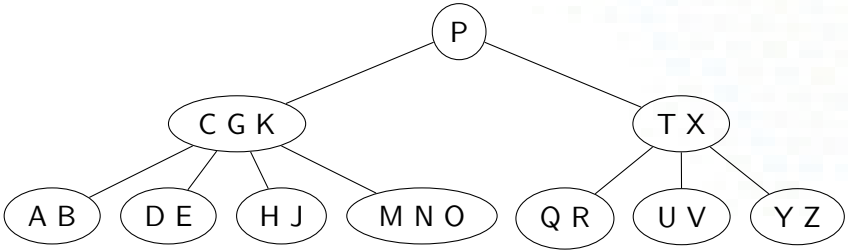
Case 2a - predecessor has $\geq t$ keys - delete L



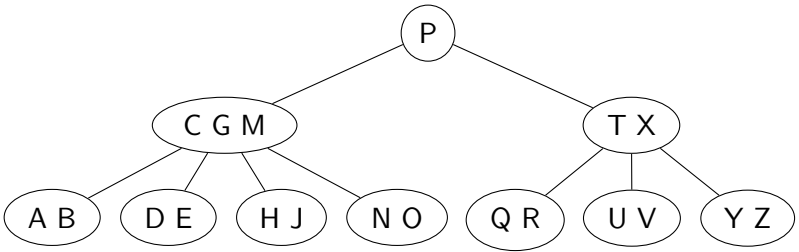
Replace L by its inorder predecessor, and recursively delete the predecessor:



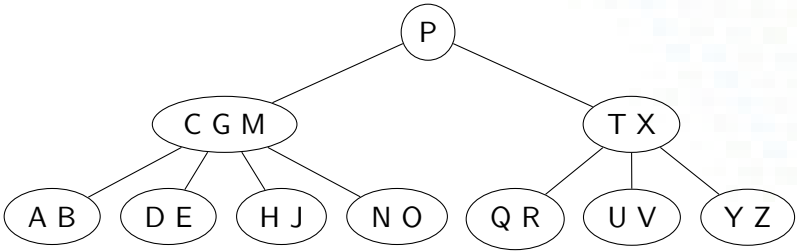
Case 2b - successor has $\geq t$ keys - delete K



Borrow from the successor:

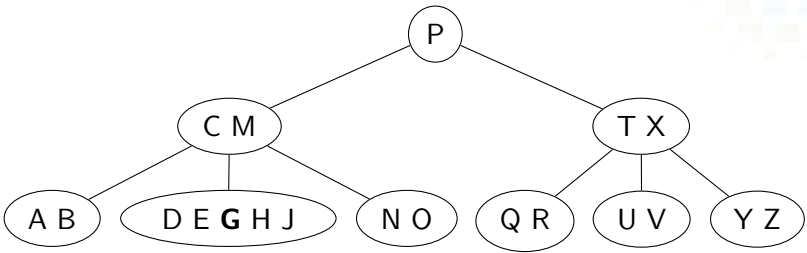


Case 2c - delete G



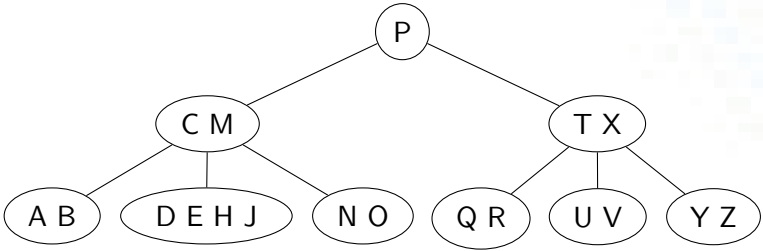
Case 2c - delete G

Neither the predecessor, nor the successor child have enough keys. In this case, we merge the two children and push G down into the new child. Then recursively delete G (which could involve any of the three cases):

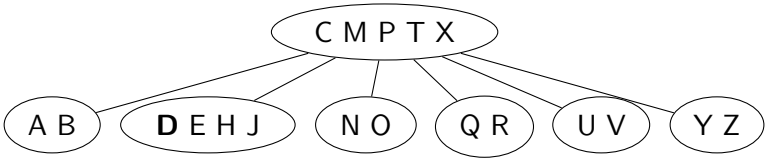


Case 3c - delete D

x is the node (C M), c is the node (D E H J):

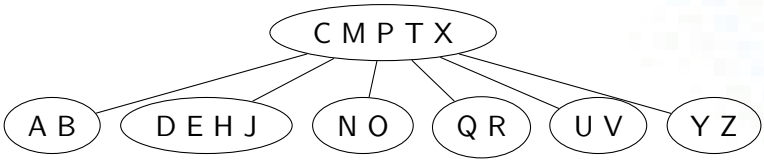


x has only one sibling and it has $t - 1$ keys. Therefore we have to merge with one of the siblings then recursively delete D:

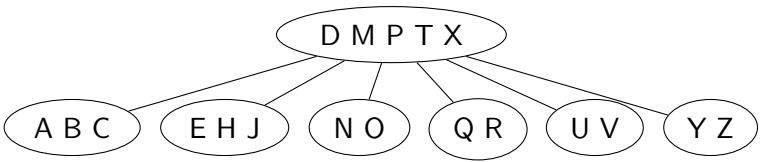


Case 3b - delete A

Let c be the node containing A.

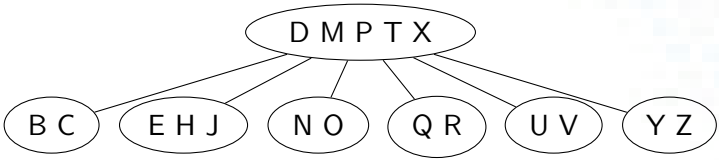


Since c's sibling has more than $t-1$ keys, we can borrow a key from its sibling:



Now we recursively delete A.

One more example - delete Z



Relevant parts of the textbook

B-trees are discussed in Chapter 18.