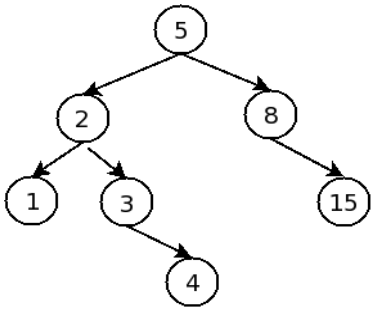
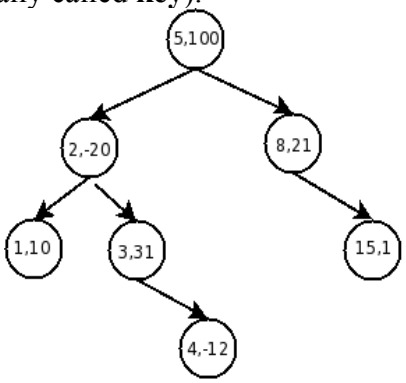
Map (associative array)
using binary tree

The default data-structure to store an array is a single chunk of memory (like C array). Contiguous memory allows fast access to data by calculating the offset from index (in C/C++, for example, a[i] = a+i*sizeof(a[0])).
The only problem with contiguous memory is that we may be wasting a lot of memory. For example, we have an array with mostly 0's in it, and a single 1 at position 10,000. To store this data in traditional array we'll need 10000*sizeof(element) bytes. Such arrays are called sparse arrays/vectors and usually stored using linked lists (all data that is not in a linked list is assumed to be 0 – default value).

In this assignment you'll implement sparse arrays using binary trees, which allows more efficient access to data compared to linked-list implementation.
Binary tree is a sorted data-structure, where each node has at most 2 child nodes, usually called left and right, and the left child is smaller than the parent, and the right is bigger. As you can see duplicates are not allowed in this data-structure.



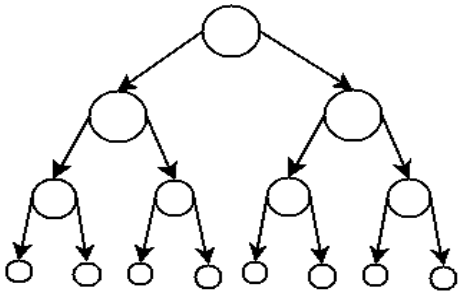
For this assignment the data in each tree node will be a pair – 2 integers, position and value. Nodes are compared by position (which is usually called **key**).



Assuming non-existing keys correspond to value 0, the above tree represents an array (0,10,-20,31,-12,100,0,0,21,0,0,0,0,0,0,1)

Standard operations on a tree are insert, delete, and find a key. All 3 may be implemented either recursively or iteratively. Recursive implementation is usually simpler, but less efficient. Both versions of **find** are already implemented, you should implement 2 version of **insert**. Insert and find are similar, they start with the root node and go down the tree till they see an appropriate child (find looks for a child with the given key/position, while insert is looking for an empty child where given value may be inserted). Let's estimate a number of key comparisons required to find a node: assuming a perfectly balanced tree – a tree where EACH node has either 2 or 0 children, and all nodes with 0 children are on the same LEVEL (denote it by L), such tree has 2^L-1 nodes. See example below with L=4, 2^L-1=15

It's obvious that the maximum number of key comparisons to find any key is L. So in order to find a node among 2^L-1 nodes we need L comparisons, or, by substituting variable 2^L=N: to find a node among N nodes organized into a binary tree we need logN comparisons. Notice that in a linked list list this operation is linear: to find a node among N nodes organized into a linked list we need N comparisons. And: to find a node among N nodes organized into an array we need ZERO comparisons.

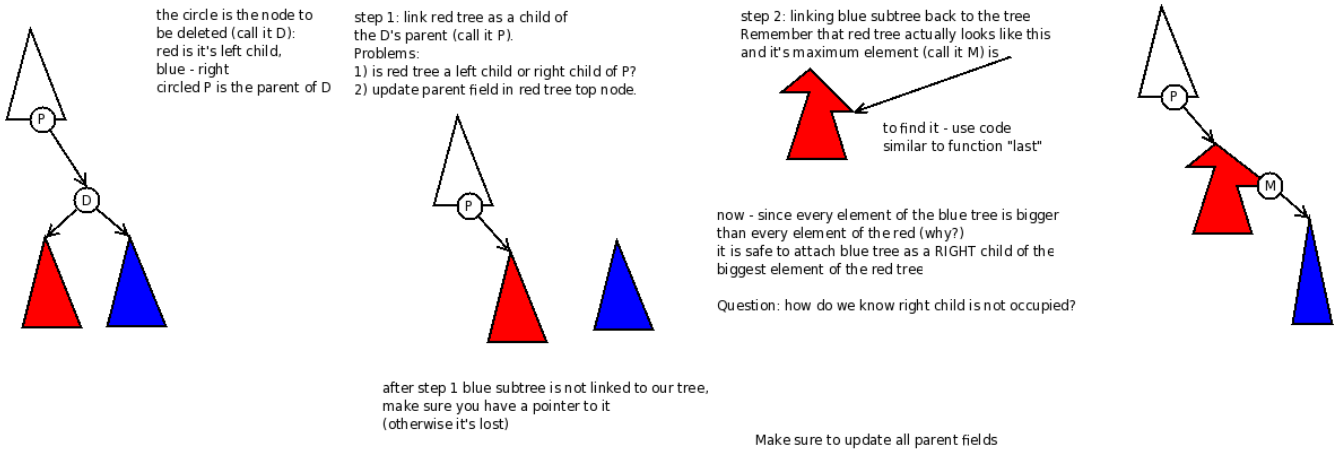


Before implementing deletion make sure you are done with decrement and last (your code will be very similar to increment and first), make sure you understand how they work. Draw pictures!

Delete is the most complex operation. There are at least 3 cases (you may need deletion of root node to be a special case as well):

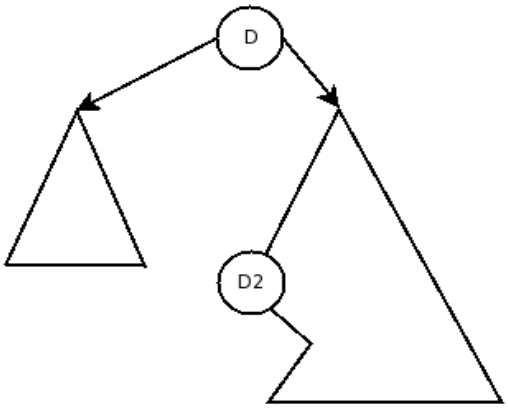
- **Deleting a node with 0 children (leaf):** just remove it from the tree, remember to update left or right pointers on the parent.
- **Deleting a node with 1 child:** delete it and replace it with its (only) child, make sure to properly update all parent fields.
- **Deleting a node with 2 children:** this is the most complex case. We'll implement 3 different versions: I've implemented a **delete_silly**, which reinserts nodes from both left and right sub-trees of the deleted node. It's 1) very inefficient and 2) it unbalances the tree (makes it look almost like a linked list, thus making subsequent finds very inefficient.

  A slightly more efficient implementation of delete is an iterative procedure described by this figure. It should be implemented in **delete(...).** Here is a diagram for deleting D which is the right child of it's parent (deleting the left child is a mirror copy – vertically flipped).

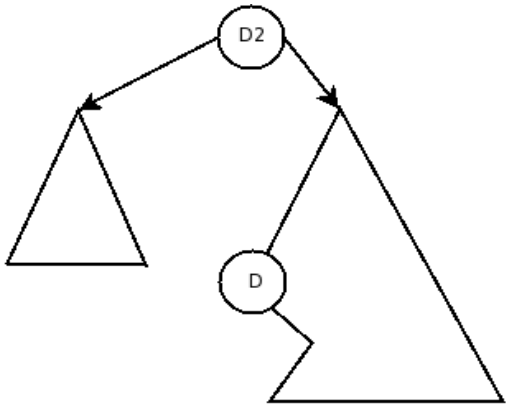


A proper implementation of delete is **delete_proper(...),** which works like this:

first **delete_proper(...) -- implemented** – finds a node to be deleted and calls **delete_proper_helper(...),** which implements cases with 0 and 1 child similar to the previous function, but case with 2 children is handled differently:



say we want to delete node D with 2 children.
Step 1) is to find a successor of D - call it D2.
Use increment function to do that. Notice that D2 is the immediate successor of D - there are no values between D and D2. The consequence of that is that D2 DOES NOT HAVE the left child !!!!!!

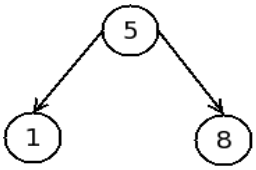


now swap the values in D and D2 (both positions and values) and delete D from it's NEW location (by calling delete_proper_helper again).
Notice that this call will
be the only recursive call: since we know that NEW node D has strictly less then 2 children, there will be no more calls.

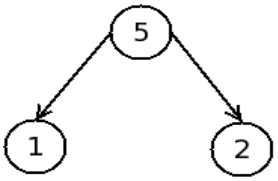
Notice that delete_proper(...) implementation will be easier if you overwrite/change the key and value fields, you still need to update some of the left,right, and parent fields.

Safety:

first of all notice that if client is able to modify the key then we have a problem. Say we created a binary tree with keys 1,5,8, which may look like



then client modifies key 8 to 2, and the tree becomes



since right child (2) IS not less than 5, this tree is not a binary tree anymore. One of the consequences is that our find function will fail to find 2 in it (why?).

So we have to be very careful:
solution 1:
add “const” to all Nodes in the code. It will work, but then how do we modify values? Remember that tree is ordered by keys, while values may be arbitrary, and one should be able to modify them freely.
We may try to add a function
void setvalue(Node* pNode, int newvalue) { pNode->value = newvalue; }
which will be useless, since all our nodes are constant (make sure you understand precisely the above situation).
One may modify setvalue, to use a cast:
void setvalue(const Node* pNode, int newvalue) { ((Node*)pNode)->value = newvalue; }
but this is UGLY, and if you can do this, then the client also can do it, and we are back to our problem.

So let's step back and ask ourselves what we are trying to achieve:

client cannot do "pNode->key = newkey;"

we tried making "pNode->" illegal by making pNode a constant, but that didn't work. Then let's make "**->key"** part ILLEGAL for the client. The ONLY way to achieve that is to HIDE struct Node definition from the client. This is called:

Opaque pointers.
Step 1: move struct Node definition into map.c (thus it is hidden from the client)
Step 2: in map.h add
typedef struct Node* Node_handle;
note that when compiler reads the above line, it DOES NOT know the definition of Node – map.c has not been seen yet (make sure you understand why). Therefore Node is an **incomplete type.** The rule in C/C++ is that **pointers to incomplete types are legal** (but dereferencing of those pointers is **not**)
Step 3: rewrite all function to use Node_handle instead of Node* and Node_handle* instead of Node**.
Note that **inside** map.c you may use Node type and Node structure fields, since by that time compiler will have full information about Node structure. **Note** that **inside** map.h you may NOT use Node* and Node_handle interchangeably.
Example: this is legal in map.c, but not map.h and or driver.c:
Node p = ...;
p.key = 5;

So what?
as we mentioned, since "->key" is illegal in the driver, client cannot modify keys and binary tree is safe. To make modification of value possible, implement setvalue(...) function. Notice how much similar opaque pointers to private data in a C++ class!

Opaque pointers are more than just safety. They also provide encapsulation. Here is a typical example: in a big project we have a struct Node in a file map.h – so it's definition is visible to everyone. Node uses "left" and "right" names for fields. Since "left" and "right" name are visible outside of the class (like public data in a C++ class) other files may use them (and does that, even though methods like getleft() and getright() are implemented). Then the maintainer of map.h decides to change the names to "Left" and "Right" (or may be change the types or eliminate them completely). Now **every** file that uses map.h is broken and has to be updated, which may be a lot of work. Opaque pointers help to avoid this kind of problems by FORCING the clients to use available getters and setters (getleft() and getright()) instead of directly accessing the data. So now the change will proceed like this:

- implementor changes left to Left
- updates getleft() from
  ... getleft() { return left; }
  to
  ... getleft() { return Left; }
- recompiles the project
- that's all, no other code is effected since everyone was using getleft() instead of left.

Notes:
1) the only reason to have field “parent” is to implement increment and decrement functions. BUT IT WORTH IT!!!
increment and decrement allow me efficient (without recursion) traversal of the tree. I used it a lot in various print_*_padded functions.
(OK – there is one more reason – to make student's life more interesting)
2) tree traversal (recursive and iterative) are very popular interview questions. Also take a look at permute function – another interview question, not that popular, since it's quite difficult.

To submit: map.c and index.chm