Lecture 5: Linked Structures

Maintaining a Sorted List of Values:

Suppose we have a stream of numbers coming in and we want to keep them sorted. We might want to quickly return the minimum, maximum or median.

We could do this:

```python
tlist.append(val)
tlist.sort()
```

Each time we get a new value val. This is horribly slow! `append` is fast and runs in $O(1)$ time, but `sort()` takes $O(n \log n)$ time where $n$ is the size of the Python list. If we have thousands of numbers, we are running `sort()` thousands of times.

Binary Search:

We do not want to rerun sorting on the entire list when we add a number to the list. Using binary search, also called bisection search, we can figure out where exactly to add the number to a sorted list in $O(\log n)$ time which is much faster than scanning the list which takes $O(n)$ time.

Here is how binary search works. We assume a sorted list. Let’s assume that we wish to find whether a number val exists in the list or not. We check the middle element of the list. If the middle element is less than val, we only have to look at the second (or right) half of the list. If the middle element is greater than val, we only have to look at the first (or left) half of the list. If the middle element equals val, we are done, and we can return `True`. Note that at each step, we are looking at a sub-list that is half the size of the original list. If we end up with one element in the list and val is not equal to this element, we can return `False`.

In our application, we want to insert the number in a sorted list in such a way that it remains sorted. The binary search procedure above needs to be modified slightly: we return the index of the number to the right of which the number val needs to be inserted. The code for this is shown below and is in `binarysearch.py`.

1 Remember that a Python List is actually an array that can be randomly accessed efficiently.
def bisect_right(a, x, lo=0, hi=None):
    """Return the index where to insert item x in list a, assuming a is sorted."
    if lo < 0:
        raise ValueError('lo must be non-negative')
    if hi is None:
        hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]:
            hi = mid
        else:
            lo = mid+1
    return lo

The return value i is such that all e in a[:i] have e <= x, and all e in a[i:] have e > x. So if x already appears in the list, a.insert(x) will insert just after the rightmost x already there.

This means that we can solve our problem using:

    a.insert(bisect_right(a, x), x)

for each number x. This is much faster than our first attempt of repeated sorting, but still not particularly fast. The issue is not with bisect_right(), which finds the insertion index very fast but with insert. Unfortunately, because a Python list is implemented as an array, inserting an element into the middle of an array requires $O(n)$ time since the elements to the right of the insertion index (returned by bisect_right(a, x) above) all have to be copied to new memory locations.

How can we speed things up? We need a linked structure where we can insert a new element using a constant number of operations, i.e., in $O(1)$ time. Consider a Linked List shown below which is initially 17, 26, 54, 77 and 93. Imagine a dictionary that represents the Linked List:

    linkedl = { "head": [17, "A"],
                "A": [26, "B"],
                "B": [54, "C"],
                "C": [77, "D"],
                "D": [93, None] }

linkedl["head"] gives us the first element of the list, and we can find the number it stores, namely, linkedl["head"][0], and find the second element using second = linkedl["head"][1], and linkedl[second].
We wish to add 31, and since we want to maintain order, we add 31 in between 26 and 54. This is shown pictorially below.

![Linked List Diagram](image)

The new Linked List should look like this:

```python
linkedl = {
    "head": [17, "A"],
    "A": [26, "T"],
    "B": [54, "C"],
    "C": [77, "D"],
    "D": [93, None],
    "T": [31, "B"]
}
```

The details are not important, but this can be done in $O(1)$ time, i.e., a constant number of operations once we have determined where to insert. We merely need to set `linkedl[“A”][1] = “T”`, and set `linkedl[“T”] = [31, “B”]` – the former modifies a number corresponding to the value of an existing dictionary key, and the latter adds a key/value pair to the dictionary representation. We have solved the problem of slow insert into a Python List, but of course, we are left with the problem of $O(n)$ search to determine where to insert 31. We cannot index into the Linked List like we could with a Python List. Binary Search Trees to the rescue!

**Binary Search Trees:**

Binary Search Trees (BSTs) allow for efficient binary search like Python lists, but also allow for efficient insert. Search and insert each take $O(h)$ time, where $h$ is the depth of the BST. Typically, $h$ grows as $\log n$, though guaranteeing this is not easy and will be a notion left for 6.006.

A binary search tree is a rooted binary tree, whose internal nodes correspond to keys with associated numbers and each key has two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the *binary search tree property*, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree. The leaves (final nodes) of the tree contain no key and have no structure to distinguish them
from one another. Leaves are commonly represented by a special leaf symbol, None. An example is shown below – the leaves are not shown.

We will use a dictionary representation for a BST similar to our Linked List dictionary representation except that each value for the key will be a triple, corresponding to a number, and a left child and a right child, in that order. Here's the representation for the BST above:

```
bst = { "root": [31, "A", "B"],
       "A": [16, "C", "D"],
       "B": [45, None, None],
       "C": [7, None, None],
       "D": [24, None, None] }
```

We will describe search, insert and delete pictorially below. Below we are searching the BST for an element with number 3.² The purple arrows indicate the steps of the algorithm.

---
² From Encrypt3D.
Below we show how a new node with number 4 is inserted into the BST. The node is inserted based on the number associated with the node, since we have to maintain the BST property. We search for 4 and fail to find it, but in the process we figure out where to insert it.

Finally, here is an example of deleting from a BST. Deletion is significantly more complicated than insert, and we show an example of deleting a node with a single child below.

---

3 From Codeward Bound.
4 From Victor Adamchik.
If a node has two children and needs to be deleted, the process is more involved. Below is a nice description of various cases in delete.\(^5\)

### Binary search tree: Deleting a node

The delete operation on binary search tree is more complicated, than insert and search. It can be divided into two stages:

- search for a node to delete;
- if the node is found, run the delete algorithm.

Here’s a more detailed description of a delete algorithm. The first stage is identical to the algorithm for search except we should track the parent of the current node. The second part is trickier. There are three cases, which are described below.

1. Node to be deleted has no children.

   This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

   **Example.** Delete -4 from a BST.

\(^5\)[http://www.algolist.net/Data_structures/Binary_search_tree/Removal]
2. Node to be deleted has one child. 

In this case, node is cut from the tree and algorithm links single child (with its subtree) directly to the parent of the deleted node.

Example. Delete 18 from a BST.
3. Node to be deleted has two children.

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example the BSTs below:

```
5
/   \/
21  19 25
|    |
19   5
```

contain the same values \{5, 19, 21, 25\}. To transform first tree into second one, we can do following:

- choose minimum element from the right subtree (19 in the example);
- replace 5 by 19;
- hang 5 as a left child.

The same approach can be utilized to delete a node, which has two children:

- find a minimum value in the right subtree;
- replace value of the node to be deleted with found minimum. Now, the right subtree contains a duplicate!
- apply delete to the right subtree to delete a duplicate.

Notice, that the node with minimum value has no left child and, therefore, its removal may result in first or second cases only.

**Example.** Delete 12 from a BST.
Find minimum element in the right subtree of the node to be deleted. In current example it is 19. Then, replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.

Now delete 19 from the subtree.
Code for BST search, insert and delete, can be found in *BST-dict-code.py*.