Lecture 2 Data Structures

1. Static array: consecutive memory; and in time.
2. Dynamic sequence
   1. Linked list: each object contains an item and a pointer to next



* 1. Python list: pre-allocate more memory than number of items in list; re-allocate memory only when necessary (no enough space, or too low fill ratio); insert/delete at last in amortized time.

1. Operation time complexity

|  |  |  |  |
| --- | --- | --- | --- |
|  | Static array | Linked list | Python list |
| get\_at  set\_at | O(1) | O(n) | O(1) |
| insert\_first  delete\_first | O(n)  reallocate memory | O(1) | O(n)  Shift elements |
| insert\_last  delete\_last | O(n)  reallocate memory | O(n) for single link  O(1) for double link | O(1) amortized |
| insert\_at  delete\_at | O(n) | O(n) | O(n) |

Problem Session 1

1. Can we have a data structure that support worst-case time index look up, as well as amortized time insertion and removal at both ends?

Lecture 3 Sets and Sorting

1. A sorted array supports find in log(n) time, find\_min and find\_max in constant time. We could use sorted array to implement a set interface.
2. Merge sort: implement in-place merge sort

Lecture 4 Hashing

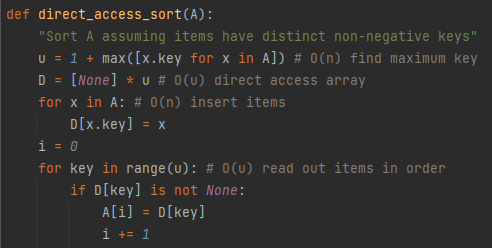
1. Find operation is lower bounded by time for a comparison model. Proof(?): A binary tree containing n nodes has depth of , and the length of a branch is the number of operations required.
2. Idea of hashing
   1. How can we support find operation with better than time? An idea is to use key as index: given integer key store items in an array at index .
   2. However, space complexity is , have to use too much space if .
   3. Hash function , where is length of the array to store the items. Store item with key k at index
3. Hash collision: since , there are always different keys such that . The idea of hashing will make us store different items at the same index. How to solve this problem?
   1. Solution 1: open addressing. Store somewhere else in the array. Complicated analysis but common and practical.
   2. Solution 2: chaining. At each index, store collision items in a “chain” (which can be implemented as array/sorted array/linked list/etc.). When need to find an item, just traverse the chain.
4. Hash function
   1. Division (bad choice):
      1. Good when keys are uniformly distributed
      2. But for some input set, will create a size chain
      3. Idea: do not use a fixed hash function; choose one randomly (but carefully)
   2. Universal (good theoretically):
      1. Hash family , parametrized by a fixed prime
      2. Each chain has size in expectation.

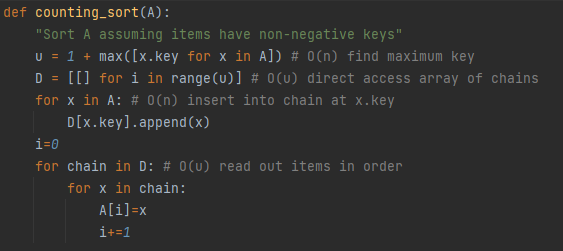
Lecture 5 Linear Sort

1. In a comparison model, the lower bound for sorting is . The argument is similar to search: sort is to search for a particular permutation, thus
2. Direct access array sort/counting sort: suppose all keys are unique non-negative integers in range . Then we insert each item into a direct access array of size . Return the items in the order in the array.

Time complexity:

Space complexity:





1. Tuple sort: when but , we can represent each key with a tuple such that . Then instead of maintaining an array of length , we can maintain an array of length storing tuple of .

Then to sort the origin keys, we can apply counting sort first by then by (from least significant to most significant). Given the sorting is stable, the original keys are sorted. Similar to sort a table by multiple columns.

1. Radix sort: when is large, we can find some constant that . Then we can represent each key in with a tuple of c elements. Using counting sort, we sort each element in the tuple, from least significant to most significant.

Need to ensure the sort for each element is stable.

The total time is . When c is constant, it is sorting.

Lecture 6 Binary Trees I

1. By now, all the data structures we have seen need linear time for either inserting/deleting or find\_min/find\_max. Binary tree can do in time.
2. Each node in a binary tree has a parent, a left child and a right child.
3. BST property: for every node, every key in left subtree <= node’s key <= every key in right subtree.
4. The following operations can be done in time. For details see lecture notes and recitation:
   1. Find the first/last node in a tree
   2. Find successor/predecessor of a node
   3. Insert a node before/after a node
   4. Remove a node
5. Binary tree can be used to implement set, which supports find operation in time.
6. Binary tree can also be used to implement sequence; but in order to support get operation, we need to maintain size of each node’s subtree.

Lecture 7 Binary Trees II: AVL

1. In lecture 6, we know binary trees can achieve lots of operations in time. The next goal is to maintain , and then the operations will be in time.
2. Rotation can (1) change the structure of a binary tree (2) preserve traversal order (3) in time
3. Theorem: rotations can transform a binary tree to any other with the same traversal order.

Proof: Any tree can be transformed into a chain in at most n-1 right rotations. Reverse the rotations to target tree.

1. AVL tree: maintain height balance: height of left and right subtrees differs by at most 1.
2. How to maintain height rebalance?
   1. Define skew = right.height – left.height.
   2. Local rebalance: for a node with skew = 2, and every other subtrees height-rebalance, can be rebalanced via 1 or 2 rotations. Details see lecture notes.
   3. Global rebalance: when adding/removing a node from a balanced tree, only the ancestors have height affected. There are ancestors, so only need to rebalance nodes.
   4. How to get the height of each node efficiently? Augment each node with height.
3. Binary tree augmentation.
   1. Add a subtree property in each node.
   2. The subtree property should be computed from children’s subtree property in time
   3. Example: in AVL tree, augment height; in binary tree sequence, augment size
4. How to build an AVL tree from a sorted array in time?

Set the median as root, and then recursively build left and right subtrees.

Lecture 8 Binary Heaps

1. Complete binary tree: every level except the deepest is fully filled, and the deepest level is left aligned.
2. Complete binary trees can be implicated represented by an array.
3. Heap property: in a max heap, for any given node C,
4. Insert node in heap time: (1) append as the last node; (2) compare with parent. If , then swap and . (3) recursively apply (2) to parent until it satisfies heap property.
5. Remove max from heap time: (1) swap root with last item and delete (2) compare current root with its children. If less than any children, then swap root with the largest child. (3) recursively apply (2) to the node until it satisfies heap property.
6. Heap sort: build a heap from the array, and delete\_max one by one. Takes time.
7. Linear build heap
   1. If we insert n items into heap, then as insert takes time, the build operation will take time
   2. Instead, we can first treat the array as a complete binary tree (without heap property), then max\_heapify\_down(i) for from to . Then total time =