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

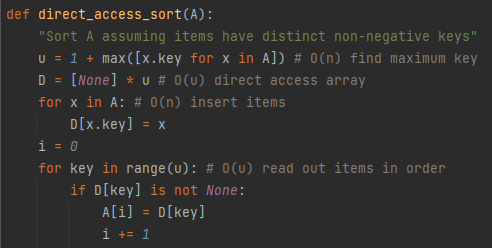
1. Find operation is lower bounded by time for a comparison model. Proof(?): A binary tree containing n nodes has depth of , and the length of a branch is the number of operations required.
2. Idea of hashing
   1. How can we support find operation with better than time? An idea is to use key as index: 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.

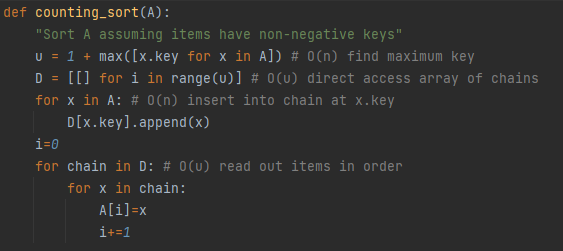
Lecture 5 Linear Sort

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 1

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.