Searching and Binary Search Trees

Richard Hua

Semester 2, 2021

Introduction

Searching for a particular item in a large database is a very common task in many different types of applications.
Similar to sorting, each item in the record is associated with a **key** that we can use in the search.
The search problem is as follows: Suppose we are a data structure D of records.

- On input of a given search key k.
- Return the record associated with k in D (a **successful search**).
- Indicated that D does not contain any record with key k (a **unsuccessful search**).
- If multiple records in D have the same key k then return any occurrence (uncommon in practice).

Continued

An **associative array** (or **dictionary**) is an ADT that relates a disjoint set of keys to an arbitrary set of values.
Keys do not need to have any ordering relation. And the items can be in any order in the dictionary (note that an actual 'dictionary' is sorted).
Also called a **table**. Named after how compilers uses a symbol table to keep track of variables of a program and their type, address, etc.

Table ADT

Definition

The **table** ADT is a set of ordered pairs of table entries of the form (k, v) where k is an unique key and v is a data value associated with the key k. Each key uniquely identifies its entry, and table entries are distinguished by keys because no two distinct entries have identical keys.
Basic operations for the table ADT include enumeration, search, insert, delete, retrieve and update.

Note that ideally we do not want to allow duplicate keys in the table ADT (unlike the searching problem in general), but this is often hard (or impossible) to do.

Continued

Table operations can be related to a function (mapping) from keys to values (e.g. $f : K \to V$).

Exercise

What type of function is f ? E.g. Is f injective? Or surjective? Or both (bijective)?

Continued

Exercise

What type of function is f ? E.g. Is f injective? Or surjective? Or both (bijective)?

f is injective since by definition no two items have identical keys and each key uniquely identifies an item.
f is surjective since all items must have an associated key.

$\therefore$ bijective

Example

Key		**Associated value** v		
Code	k	City	Country	State / Place
AKL	271	Auckland	New Zealand	
DCA	2080	Washington	USA	District of Columbia (D.C.)
FRA	3822	Frankfurt	Germany	Hesse
GLA	4342	Glasgow	UK	Scotland
HKG	4998	Hong Kong	China	
LAX	7459	Los Angeles	USA	California
SDF	12251	Louisville	USA	Kentucky
ORY	9930	Paris	France	

Note that each identifier (code) has a unique integer representation $k = 26^2 c_0 + 26 c_1 + c_2$ if we map $\{A, B, \cdots, Z\} \to \{0, 1, \cdots, 25\}$.

Continued

There are two types of table ADTs in general.

- **Static**. The database is fixed in advance and no insertions, deletions or updates etc. are allowed. (not common)
- **Dynamic**. table entries could be changed

Properties

We have not discussed about the implementations of the table ADT yet. But we do have some basic theoretical results.

Lemma

Suppose that a table is built up from empty by successive insertions, and we then search for a key k uniformly at random. Let $T_{ss}(k)$ (respectively $T_{us}(k)$) be the time to perform successful (respectively unsuccessful) search for k. Then

- *the time taken to retrieve, delete, or update an element with key k is at least $T_{ss}(k)$;*
- *the time taken to insert an element with key k is at least $T_{us}(k)$;*
- *$T_{ss}(k) \leqslant T_{us}(k)$;*
- *$T_{ss}(k) = T_{us}(k)$ in the worst case;*
- *the average value of $T_{ss}(k)$ equals one plus the average of the times for the unsuccessful searches undertaken while building the table.*

Continued

Proof.

To retrieve, delete or update an element, we need to search for that element first so $T_{ss}(k)$ is a lower bound.
Similarly, to insert an element with key k, we have to (unsuccessfully) search for it first just to make sure its slot has not been taken. Therefore, for a given state of the table formed by insertions from an empty table, the time for successful search for a given element is the time that it took for unsuccessful search for that element, as we built the table, plus one.
This means that the time for unsuccessful search is always at least the time for successful search for a given element (the same in the worst case), and the average time for successful search for an element in a table is the average of all the times for unsuccessful searches plus one.

□

Continued

Note that we didn't give any actual time complexities so far.
Actual time complexity depends on how the table ADT is implemented.
Example, if the data structure is a list, then it depends on whether the list is sorted.
Also note that we mentioned in theory that the mapping is bijective, so we should be able to do a 'reversed' searching as well.
Often only possible if a specific reverse-mapping is pre-computed.
E.g. We normally search through a telephone directory by name (key). But it is almost hopeless to search the directory for a specific name by number (value).

Sequential search

The most basic implementation is the **sequential search** which can work on both sorted and unsorted input list.
Starts at the head of the list and examines elements in order until it finds the desired key or reaches the end of the list.

function SEQUENTIALSEARCH(list $a[0 \cdots n-1]$, key k)
  for $i \leftarrow 0$ **while** $i < n$ **step** $i \leftarrow i+1$ **do**
    if $a[i] = k$ **then return** i
  return NOT FOUND

Worst and average-case time complexity for both successful and unsuccessful search is both $\Theta(n)$.

Not so efficient

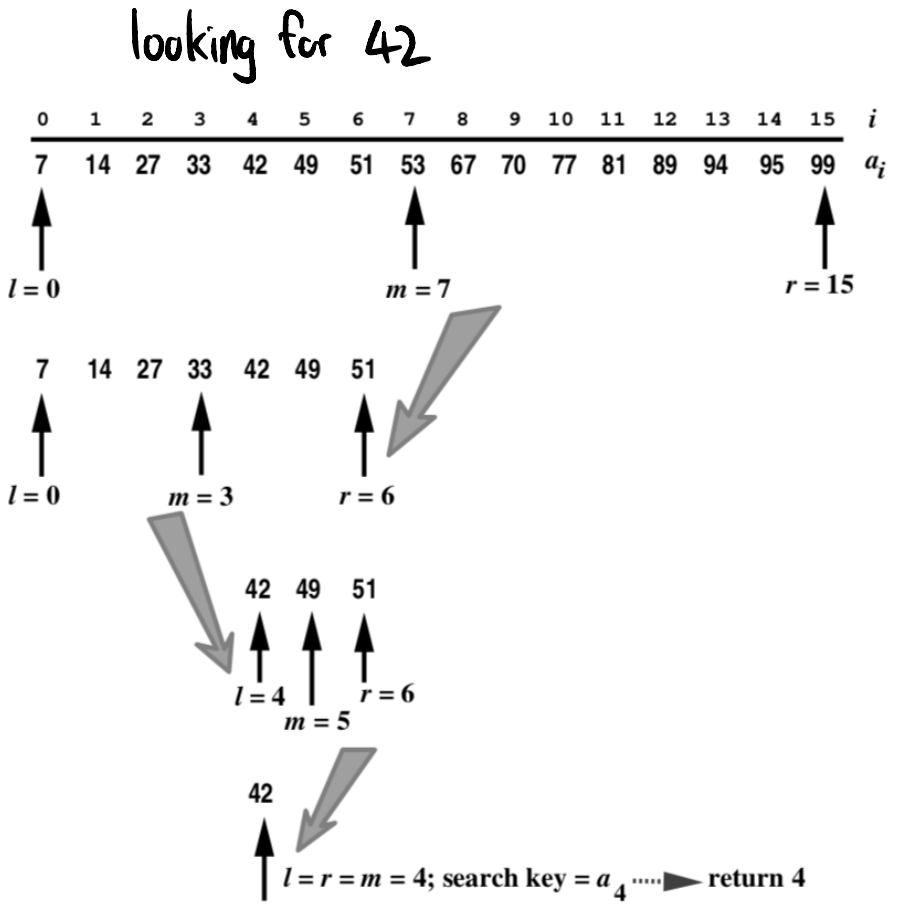
Sorted lists

A sorted list implementation allows for much better search method. The basic idea of **binary search** is as follows:

- Suppose k is the key we wish to search.
- If the list is empty, return 'not found'.
- Choose the key $a[m]$ of the middle element of the list. If $a[m] = k$, return its record; if $a[m] > k$, make a recursive call on the head sublist; if $a[m] < k$, make a recursive call on the tail sublist.

a[m] is sorted list

Example



Middle index m is calculated as $\lfloor \frac{l+r}{2} \rfloor$.

Analysis

Binary search is much more efficient on arrays than on linked-lists.

Lemma

Consider a sorted list implementation of the table ADT.
Using an array, both successful and unsuccessful search take time in $\Theta(\log n)$*, in both the average and worst case, as do retrieval and updating. Insertion and deletion take time in* $\Theta(n)$ *in the worst and average case.*
Using a linked-list, all the above operations take time in $\Theta(n)$*.*

Continued

Proof.

Unsuccessful search does $\lceil \lg n \rceil$ number of comparisons in all cases since we half the search range each time. By the previous Lemma successful search is also $\Theta(\log n)$ in the worst and average case.
Insertion and deletion take linear time in arrays since we need to copy everything to a new arrays of size $n+1$ or $n-1$.
On linked-lists, note that after we narrow down our current search range, (one of) the pointer(s) has to travel half the distance between them to get the middle element. So the total distance travel looks something like $n/2 + n/4 + \cdots \leqslant n \in \Theta(n)$. Once we find insertion/deletion point, adding or removing the node takes constant time. □

Binary search tree

Binary search performs a predetermined sequence of comparisons depending on n and k.
This sequence is easier to analyse if we represent the sorted list as a **binary search tree** (**BST**).

Definition

A **binary search tree** (BST) is a binary tree that satisfies the following ordering relation: for every node v in the tree, the values of all the keys in the left subtree are smaller than (or equal, if duplicates are allowed) to the key in v and the values of all the keys in the right subtree are greater than the key in v.

Continued

Suppose we are given a reference-based BST, all keys can be placed in sorted order by inorder traversal (won't work if duplicates are allowed).
We consider the BST representation of a sorted list.
The element at the middle index $m_0 = \lfloor (n-1)/2 \rfloor$ is the root.
The lower sublist $a[0, \cdots, m_0 - 1]$ and upper sublist $a[m_0 + 1, \cdots n - 1]$ are related to the left and right arcs from the root.
Recursively define the right and left subtrees.

Inorder traversal

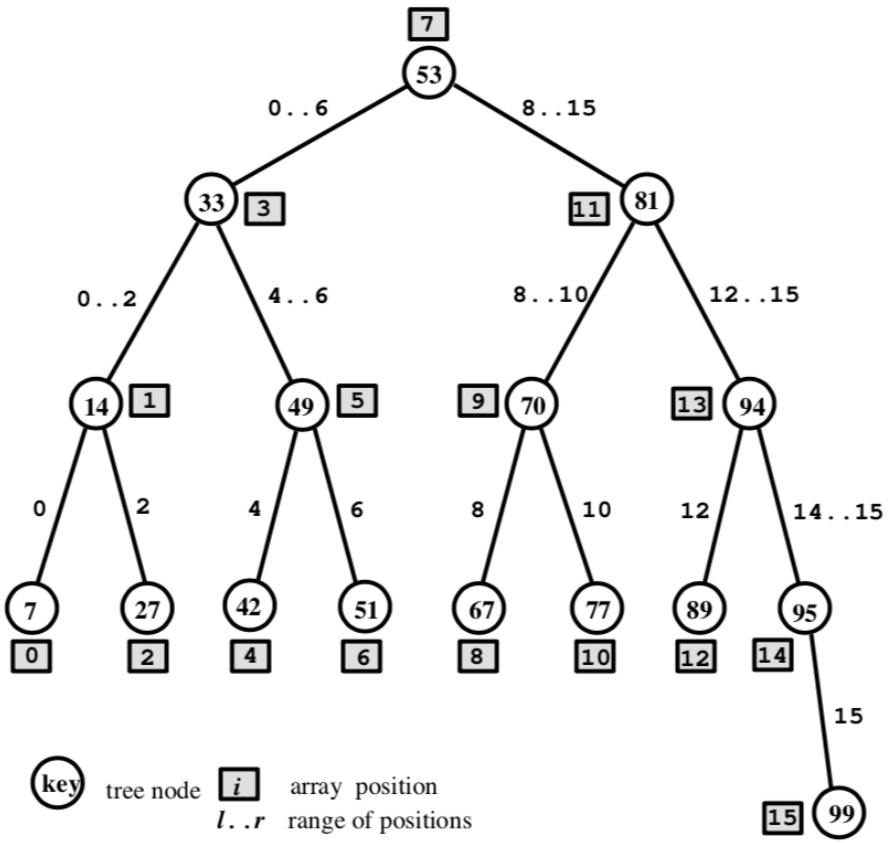
Given a binary tree T. The inorder traversal of the tree is a recursive algorithm that first recursively call itself on the left subtree of the current node, then visit the node itself and then visit the right subtree of the node recursively.
To sort all keys in the BST, we can do an inorder traversal starting at the root of the tree.

Pseudo-code

```
function INORDERTRAVERSAL(root v)
    if v = NULL then return
    else
        INORDERTRAVERSAL(v.leftChild)
        Print(v.value)
        INORDERTRAVERSAL(v.rightChild)
    return
```

It's called inorder traversal because we visit the value of v in-between the two recursive calls.
Similarly, there is also preorder and postorder traversal.

Example


7
53
0..6
8..15
33
3
11
81
0..2
4..6
8..10
12..15
14
1
49
5
9
70
13
94
0
2
4
6
8
10
12
14..15
7
27
42
51
67
77
89
95
0
2
4
6
8
10
12
14
15
99
key
tree node
i
array position
l..r
range of positions

Continued

Search can be interpreted as a path starting from the root of the tree.
Number of comparisons to find a key is equal to the number or nodes along the unique path from the root to the key (**depth** of the node, plus one).
Note that our BST here is well-balanced: for each node in the tree, the left and right subtrees have height difference of at most 1. So all leaves are on at most two levels.

Pseudo-code

function BINARYSEARCH(list $a[0 \cdots n-1]$, key k)
    $l \leftarrow 0$; $r \leftarrow n-1$
    while $l \leqslant r$ **do**
        $m \leftarrow \lfloor (l+r)/2 \rfloor$
        if $a[m] < k$ **then** $l \leftarrow m+1$
        else if $a[m] > k$ **then** $r \leftarrow m-1$
        else return m
    return NOT FOUND

This is known as 3-way comparisons binary search. We can improve the constant by using two-way comparisons.

Pseudo-code

function BINARYSEARCH2(list $a[0 \cdots n-1]$, key k)
    $l \leftarrow 0$; $r \leftarrow n-1$
    while $l < r$ **do**
        $m \leftarrow \lfloor (l+r)/2 \rfloor$
        if $a[m] < k$ **then** $l \leftarrow m+1$
        else $r \leftarrow m$
    if $a[l] = k$ **then return** m
    else return NOT FOUND

Improvement is minor and analyzing the exact difference between the two versions is quite difficult.

Continued

We have used a BST to specify sequences of comparisons of binary search.
We will use BSTs not just as a mathematical object to aid our analysis, but as an explicit data structure to implement the table ADT.