Data Searching and Binary Search

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

1. Data search problem
2. Static and dynamic search
3. Sequential search
4. Sorted lists and binary search

Data Search in a Large Database

Searching in a database D of **records**, such that each record has a **key** to use in the search.

THE SEARCH PROBLEM: Given a search key k, either

- return the record associated with k in D (a **successful search**: if k occurs several times, return any occurrence), or
- indicate that k is not found, without altering D (an **unsuccessful search**).

The purpose of the search:

- to access data in the record for processing, or
- to update information in the record, or
- to insert a new record or delete the record found.

Tables: General-Case Data Structures for Searching

An **associative array**, or **dictionary**, or a **table**:

- A key and a value are linked by *association*.
- An abstract data type (ADT) relating a disjoint set of keys to an arbitrary set of values.
- Keys of entries may not have any ordering relation and may be of unknown range.
- No upper bound on the table size: an arbitrary number of different data items can be maintained simultaneously.
- No analogy with a conventional word dictionary, having a lexicographical order.

Definition 3.1 (Textbook): The **table** ADT is a set of ordered pairs, or table entries (k, v) where k is an unique *key* and v is a data value associated with the key k.

Basic Operations for Tables

Abstractly, a table is a mapping (function) from keys to values.

Given a search key k, **table search** has to find the table entry (k, v) containing that key. After the search, one may:

- RETRIEVE the found entry (k, v), e.g., to process v;
- REMOVE, or *delete* the found entry from the table;
- UPDATE its value v;
- INSERT a new entry with key k if the table has no such entry.

Additional operations on a table:

- INITIALIZE a table to the empty one;
- INDICATE an unsuccessful search (i.e., that there is no entry with the given key).

Types of Search

- **Static search**: unalterable (fixed in advance) databases; no updates, deletions, or insertions.
- **Dynamic search**: alterable databases (allowable insertions, deletions, and updates).

Key		**Associated value** v		
Code	k	City	Country	State/Place
AKL	271	Auckland	New Zealand	North Island
DCA	2080	Washington	USA	District of Columbia (D.C.)
FRA	3822	Frankfurt	Germany	Hesse
SDF	12251	Louisville	USA	Kentucky

A unique integer key $k = 26^2 c_0 + 26 c_1 + c_2$ for 3-letter identifiers: (c_i; $i = 0, 1, 2$ – ordinal numbers of A..Z in the English alphabet: A – 0; B – 1; ..., Z – 25).

Basic implementations of the table ADT: *lists* and *trees*.

- An *elementary operation*: a query or update of a list element or tree node, or comparison of two of them.

Sequential Search in Unsorted Lists

Starting at the head of a list and examining elements one by one until finding the desired key or reaching the end of the list.

Exercise 3.1.1. Both successful and unsuccessful sequential search have worst-case and average-case time complexity $\Theta(n)$.

Proof: The unsuccessful search explores each of n keys, so the worst- and average-case time is $\Theta(n)$.

The successful search examines n keys in the worst case and $\frac{n}{2}$ keys on the average, which is still $\Theta(n)$. □

- The sequential search is the only option for unsorted arrays and linked lists of records.
- A sorted list implementation allows for much better search based on the divide-and-conquer paradigm.

Binary Search in a Sorted List L of Records

$$L = \{(k_i, v_i) : \; i = 1, \ldots, n; \; k_1 < k_2 < \ldots < k_n\}$$

Recursive binary search for the key k:

1. If the list is empty, return "not found", otherwise
2. Choose the key k_m of the middle element of the list and
   - if $k_m = k$, return its record, otherwise
   - if $k_m > k$, make a recursive call on the head sublist, otherwise
   - if $k_m < k$, make a recursive call on the tail sublist.

Iterative implementation for each sublist $(k_l, k_{l+1}, \ldots, k_r)$ of keys:

- The middle index $m = \left\lfloor \frac{l+r}{2} \right\rfloor$.
- If $k_m = k$, then return the record (k_m, v_m) and terminate iterations.
- If $k_m > k$, then $r = m - 1$.
- If $k_m < k$, then $l = m + 1$.
- If $l > r$, return *"Item not found"* and terminate iterations.

Non-recursive (Iterative) Binary Search in Array

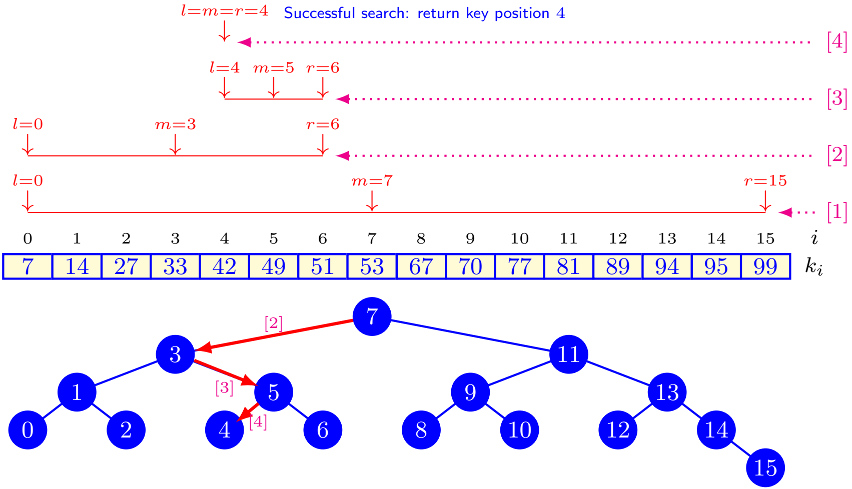
The performance of binary search on an array is much better than on a linked list because of the constant time access to a given element.

begin BinarySearch (a sorted integer array $\mathbf{k} = (k_0, k_1, \ldots, k_{n-1})$
of keys associated with items, a search key k)
  $l \leftarrow 0$; $r \leftarrow n - 1$
  while $l \leq r$ **do** $m \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$
    if $k_m < k$ **then** $l \leftarrow m + 1$
      else if $k_m > k$ **then** $r \leftarrow m - 1$
        else return m
    end if
  end while
  return *ItemNotFound*
end

Faster Binary Search with Two-way Comparisons

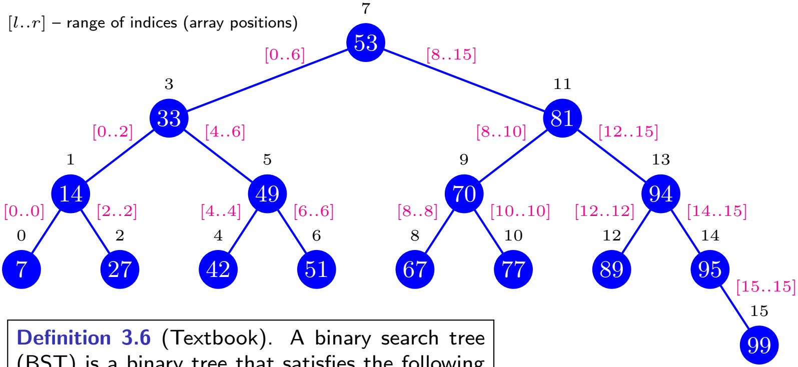
```
begin BinarySearch2 (a sorted integer array k = (k_0, k_1, ..., k_{n-1})
                     of keys associated with items, a search key k)
   l ← 0; r ← n − 1
   while l < r do m ← ⌊(l+r)/2⌋
      if k_m < k then l ← m + 1
                 else r ← m
      end if
   end while
   if k_l = k then return l
      else return ItemNotFound
   end if
end
```

Binary Search in Array $\{k_0 = 7, \ldots, k_{15} = 99\}$ for Key $k = 42$



Tree Structure of Binary Search: Binary Search Tree

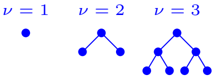
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
7	14	27	33	42	49	51	53	67	70	77	81	89	94	95	99



Definition 3.6 (Textbook). A binary search tree (BST) is a binary tree that satisfies the following ordering relation: for every node i in the tree, the values of all the keys in the left subtree are smaller than the key in i and the values of all the keys in the right subtree are greater than the key in i.

Binary Search: Worst-Case Time Complexity $\Theta(\log n)$

The complete binary tree of $n = 2^\nu - 1$ keys (each internal node has 2 children); $\nu_{\{n\}} = 1_{\{1\}}, 2_{\{3\}}, 3_{\{7\}}, \ldots$:



- The tree height is $\nu - 1$ since the tree is balanced.
- Each tree level l contains 2^l nodes:
  - $l = 0$ – the root (one node).
  - $l = 1, \ldots, \nu - 2$ – internal nodes: 2^l at each level l.
  - $l = \nu - 1$ – the $2^{\nu-1}$ leaves.
- $l + 1$ comparisons to find a key of level l (see Slide 11).
- **The worst case**: $\nu = \lg(n + 1)$ comparisons.

The worst-case time complexity of unsuccessful and successful binary search is $\Theta(\log n)$.

Binary Search: Average-Case Time Complexity $\Theta(\log n)$

Lemma: The average-case time complexity of successful and unsuccessful binary search in a balanced tree is $\Theta(\log n)$.

Proof: The depth°) of the tree is $d = \lceil \lg(n+1) \rceil - 1 \equiv \lceil \nu \rceil - 1$.

- At least half of the tree nodes have the depth at least $d-1$.
- The average depth over all nodes is at least $\frac{d}{2} \in \Theta(\log n)$.
- The average depth over all nodes of an arbitrary (not necessarily balanced) binary tree is $\Omega(\log n)$.

The expected search time for an arbitrary balanced tree is equal to the average balanced tree depth $\Theta(\log n)$. $\square$

°) Definitions (see Textbook, Appendix D7):

- Depth of a node – the length (number of edges) of the unique path to the root.
- Height of a node – the length of the longest path from the node to a leaf.
- Height of the tree – the height of the root.

Interpolation Search

Improvement of binary search if it is possible to guess where the desired key sits.

- A simple practical example: the search for C or X in a phone directory.
- Practical if the sorted keys are almost uniformly distributed over their range.
- Binary search: the middle position $m = \left\lfloor \frac{l+r}{2} \right\rfloor = l + \left\lceil \frac{r-l}{2} \right\rceil$.
- Interpolation search: the predicted position of key k if the keys are iniformly distributed between k_l and k_r:

$$m = l + \lceil \rho(r - l) \rceil \equiv l + \left\lceil \frac{k - k_l}{k_r - k_l}(r - l) \right\rceil$$

Dynamic Binary Tree Search

Static binary search is converted into a **dynamic binary tree search** by allowing for insertion and deletion of data records.

- Dynamic binary tree search makes actual use of the binary search tree (BST) data structure.
- The BST data structure is constructed by linking data records.
- A BST allows for inserting a new node.
- Any existing node of a BST may be removed.
- Using an array implementation of a sorted list, both successful and unsuccessful search, retrieval, and updating take time in $\Theta(\log n)$ on average and in the worst case.
  - But insertion and deletion are in $\Theta(n)$ in the worst and average case.
- Using a linked list, all the above operations take time in $\Theta(n)$.