TTIT33 Algorithms and Optimization – Lecture 5 Algorithms

Lecture 5 Algorithms
ADT Map, ADT Dictionary
Binary Search, Hashing, Binary Search Trees

TTIT33 – Algorithms and Optimization

Content:
- ADTs Map/Dictionary
  GT p.368-369, p.389
- Implemented as:
  - Arrays/binary search GT 9.3.3
  - Hash tables GT 9.2, 9.3.2,
  - Binary Search Trees GT 10.1.1-10.1.2

ADT Map, ADT Dictionary

- Domain: sets of pairs <k, i>
  where k key, i information,
- ADT Map: a key may appear in at most one entry!
  size, isEmpty, get(k), put(k,v), remove(k)
  Example: examination list: personal number used as key,
- ADT Dictionary: a key may appear in different entries!
  size, isEmpty, find(k), findAll(k), insert(k,v), remove(k,v).
  Example: Telephone directory (several numbers per person)

Implementations: Map, Dictionary

- Table/Array – seq. of memory chunks of equal size
  - Unordered no particular order between T[i] and T[i+1]
  - Ordered T[i] < T[i+1]
- Linked Lists
  - Unordered
  - Ordered
- Hashing [Goodrich/Tamassia 9.2]
- Binary Search Trees [Goodrich/Tamassia 10.1]

Array representations of Set and Dictionary

For unordered keys: (unordered set / dictionary)
lookUp by linear search
- unsuccessful lookup: n comparisons → O(n) time
- successful lookup, but worst case: n comparisons → O(n) time
- successful lookup, average case
  with uniform distribution of requests:
  \[ \frac{1}{n} + \frac{1}{2} + \ldots + \frac{1}{n+1} = \frac{\log(n+1)}{2} \]
  comparisons → O(n) time

Array representations of... (cont.)

For ordered keys: (ordered set / dictionary)
LookUp by binary search:

Function binLookup(table T[min..max], key k)
  if (min=max) then return NULL
  mid ← (min+max)/2,
  if k = key(T[mid])
    then return T[mid]
  else if k < key(T[mid])
    then return binLookup(T[min..mid-1])
  else return binLookup(T[mid+1..max])
Analysis of binary search

All possible executions represented by a binary tree (a binary decision tree):

- The max height of the tree is the max no of iterations of the algorithm
- Height is $O(\log(n))$
- Worst case = max un-balance, searched for key in deepest leaf...

Hash Tables

To implement ADT Map/ADT
- Use a table $T[0..m]$
- Find a (hash) function $h: key \rightarrow i \in [0..m]$ such that $k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$
- Store each key-element pair as $<k, e>$ in $T[h(k)]$
- Commonly used:
  $h(k) = k \mod m$

[GT] terminology: $h(k) = g(hc(k))$

hash code $hc$: Keys $\rightarrow$ Int; compression function $g$: Int $\rightarrow [0..m]$

Hash Table – Collision Resolution

Two principles for handling collisions:

1. Chaining: keep conflicting data in linked lists
   - Separate Chaining: Keep a linked list of the colliding ones outside the table
   - Coalesced Chaining: Store all items inside the table
2. Open Addressing: Store all items inside the table, and the index to use at collision is determined by an algorithm

Hashing with Separate Chaining: Example

- Hash table of size: 13
- Hash function $h$ with $h(k) = k \mod 13$
- Store 10 integer keys: 54, 10, 18, 25, 28, 41, 38, 36, 12, 90

Hashing with Separate Chaining: LookUp

Given: key $k$, hash table $T$, hash function $h$

1. compute $h(k)$
2. search for $k$ in the list pointed by $T[h(k)]$

Notation: probe = one access to the linked list data structure
- 1 probe for accessing the list header (if nonempty)
- $1+i$ probes for accessing the contents of the first element
- $1+2$ probes for accessing the contents of the second element
- ...

A probe (just pointer dereferencing) takes constant time. How many probes $P$ are needed to retrieve a hash table entry?
Separate Chaining: Unsuccessful LookUp

- \( n \) data items
- \( m \) positions in the table

Worst case:
- all items have the same hash value: \( P = 1 + \alpha \)

Average case:
- Hash values equally distributed among \( m \):
- Average length \( \alpha \) of the list: \( \alpha = n/m \)
- \( \alpha = n/m \) is called the load factor

\[ P = 1 + \alpha \]

Separate Chaining: Successful LookUp

Average case: Expected number \( P \) of probes for given key \( k \):

- Access to \( T[h(k)] \) (beginning of a list \( L \)): 1
- Traversing \( L \rightarrow k \) found after: \( (|L| + 1)/2 \)
- Expected (or average) \( |L| \)

\( \alpha = n/m \)

\[ P = \alpha/2 + 3/2 \]

Coalesced Chaining: items inside table (1)

First step:
- store first element in table
- keep rest in separate lists

The increase in space consumption is acceptable
if key fields are small or hash table is quite full

Coalesced Chaining: items inside table (2)

Place data items in the table
- Extend them with pointers
- Resolve collisions by using the first free slot

Chains may contain keys with different hash values...
...but all keys with the same hash value appear in the same chain
+ better space utilization
- table may become full

Open Addressing

- Store all elements inside the table
- Use a fix algorithm to find a free slot

Sequential / Linear Probing
- desired hash index \( j = h(k) \)
- in case of conflict go to the next free position
- If at end of the table, go to the beginning...

- Close positions rapidly filled (primary clustering)
- How to remove(\( k \)) ??

Open Addressing – how to remove

The element to remove may be part of a collision chain:
- cant be directly removed!
- Two approaches:
  - Scan elements after, re-hash
  - Mark as “deleted” (if the next slot is non-empty)
Double Hashing

- Second hashing function $h_2$ computes increments in case of conflicts
- Increment beyond the table is taken modulo $m = \text{tableSize}$

Linear probing is double hashing with $h_2(k) = 1$

Requirements on $h_2$:
- $h_2(k) \neq 0 \ldots$ for all $k$
- For each $k$, $h_2(k)$ has no common divisor with $m$ → all table positions can be reached

A common choice: $h_2(k) = q - (k \mod q)$ for $q < m$ prime, $m$ prime (i.e., pick a prime less than the table size!)

Binary Search Tree

A binary search tree (BST) is a binary tree such that:
- Information associated with a node includes a key, linear ordering of nodes determined by keys.
- The key of each node is:
  - greater than (or equal) the keys of all left descendents, and
  - smaller than (or equal) the keys of all right descendents.
- The leaves do not store any information

Variation in [Goodrich & Tamassia]

ADT Map as Binary Search Tree...

- lookUp($k, v$): comparison controlled traversal
  - if $key(v) = k$
  - then return $k$
  - else if $k < key(v)$ then LookUp($k, \text{leftChild}(v)$)
  - else LookUp($k, \text{rightChild}(v)$)
- worst-case: height($T$)+1 comparisons

put($k, v$): add ($k, v$) as new leaf on lookUp failure or update the node on lookUp success
- worst-case: height($T$)+1 comparisons

remove($k$): lookUp, then...
- if $v$ is a leaf, remove $v$
- if $v$ has one child $u$, replace $v$ by $u$
- if $v$ has two children, replace $v$ by its inorder successor (alternatively: by its inorder predecessor)
- worst-case: height($T$)+1 comparisons

Successful LookUp: Worst and Average case

Worst Case BST
- BST degenerated to a linear sequence
- expected number of comparisons is $(n+1)/2$

Balanced BST
- The depths of leaves differ by at most 1
- $O(\log_2 n)$ comparisons.