Lecture 3

Dictionaries and Hash Tables

Ordered Dictionaries and Binary Search Trees

AVL Trees
Lecture outline

- Dictionaries (the unordered dictionary ADT);
- Hash tables (bucket arrays, hash functions);
- Collision handling schemes;
- Sorted tables;
- Binary search trees
- Searching, inserting and removal with binary search trees;
- Performance of binary search trees;
- AVL trees (update operations, performance);
Dictionaries and Hash Tables

- A computer dictionary is a data repository designed to effectively perform the search operation. The user assigns keys to data elements and use them to search or add/remove elements.

- The dictionary ADT has methods for the insertion, removal and searching of elements.

- We store key-element pairs \((k,e)\) called **items** into a dictionary.

- In a student database (containing student’s name, address and course choices, for example) a key can be student's ID.

- There are two types of dictionaries:
  - Unordered dictionaries;
  - Ordered dictionaries;
Unordered dictionaries

In the most general case we can allow multiple elements to have the same key. However, in some applications this should not be advantageous (e.g. in a student database it would be confusing to have several different students with the same ID).

In the cases when keys are unique, a key associated to an object can be regarded as an address of this object. Such dictionaries are referred to as associative stores.

As an ADT a dictionary $D$ supports the following methods:

- **findElement($k$)** – if $D$ contains an item with key $k$, return an element $e$ of such an item, else return an exception NO_SUCH_KEY.
- **insertItem($k, e$)** – insert an item with element $e$ and key $k$ in $D$
- **RemoveElement($k$)** – remove from $D$ an item with key $k$ and return its element, else return an exception NO_SUCH_KEY.
Log Files

- One simple way of realising an unordered dictionary is by an unsorted sequence $S$, implemented itself by a vector or list that store (such implementation is referred to as a log file or audit trial).

- This implementation is suitable for storing a small amount of data that do not change much over time.

- With such implementation, the asymptotic execution times for dictionary methods are:
  - insertItem – is implemented via insertLast method on $S$ in $O(1)$ time;
  - findElement – is performed by scanning the entire sequence which takes $O(n)$ in the worst case;
  - removeElement – is performed by searching the whole sequence until the element with the key $k$ is found, again the worst case is $O(n)$;
Hash Tables

- If the keys represent the “addresses” of the elements, an effective way of implementing a dictionary is to use a hash table.

- The main components of a hash table are the bucket arrays and the hash functions.

- A bucket array for a hash table is an array $A$ of size $N$, where each “element” of $A$ is a container of key-element pairs, and $N$ is the capacity of the array.

- An element $e$ with the key $k$ is inserted into the bucket $A[k]$.

- If keys are not unique, two different elements may be mapped to the same bucket in $A$ (a collision has occurred).
Hash Tables

- There are two major drawbacks of the proposed concept:
  - The space $O(N)$ is not necessarily related to the number of items $n$ (if $N$ is much greater than $n$, considerable amount of space is unused);
  - The bucket array requires the keys to be unique integers in the range $[0,n-1]$, which is not always the case;

- To address these issues, the hash table data structure has to be defined as a bucket array, together with a good mapping from the keys to the range $[0,n-1]$;
A hash function $h$ maps each key $k$ from a dictionary to an integer in range $[0,N-1]$, where $N$ is the bucket capacity.

Instead of using $k$ as an index of the bucket array, we use $h(k)$, i.e. we store the item $(k,e)$ in the bucket $A[h(k)]$.

A hash function is deemed to be “good” if it minimises collisions as much as possible, and at the same time evaluating $h(k)$ is not expensive.

The evaluation of a hash function consists of two phases:

- Mapping $k$ to an integer (the hash code);
- Mapping the hash code to be an integer within the range of indices in a bucket array (the compression map);
Hash Functions

$k$

hash code

-2 -1 0 1 2

compression map

0 1 N-1
Hash Codes

- The first action performed on an (arbitrary) key $k$ is to assign to it an integer value (called the **hash code** or **hash value**). This integer does not have to be in the range $[0, N-1]$, but the algorithm for assigning it should avoid collisions.

- The range of values of $k$ can be larger than that assumed in the hash code (e.g. $k$ is a long integer and the hash value is a short integer).

- One way of creating a hash code in such situation is to take only lower or higher half of $k$ as a hash code.

- Another option is to sum lower and upper half of $k$ (the summation hash code).
Polynomial Hash Codes

- The summation hash code is not a good choice for keys that are character strings or other multiple-length objects of the form \((x_0, x_1, \ldots, x_{k-1})\).

- An example: Assume that \(x_i\)'s are ASCII characters and that the hash code is defined as

\[
h(k) = \sum_{i=0}^{k-1} \text{ASCII}(x_i)
\]

Such approach would produce many unwanted collisions, such as:

- temp 01
tops
- temp 10
pots
- stop
spot
Polynomial Hash Codes

- A better hash code would take into account both the values $x_i$ and their positions $i$. By choosing a constant $a$ ($a \neq 0, 1$), we can define the hash code:

$$h = x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

This is a polynomial in $a$ with the coefficients $(x_0, x_1, \ldots, x_{k-1})$.

- Hence, this is the polynomial hash code.

- Experimental studies show that good spreads of hash codes are obtained with certain choices for $a$ (for example, 33, 37, 39, 41);

- Tested on the case of 50,000 English words, each of these choices provided fewer than 7 collisions.
Compression Maps

- If the range of hash codes generated for the keys exceeds the index range of the bucket array, the attempt to write the element would cause an out-of-bounds exception (either because the index is negative, or out of range \([0,N-1]\). 

- The process of mapping an arbitrary integer to a range \([0,N-1]\) is called compression, and is the second step in evaluating the hash function.

- A compression map that uses:
  \[
  h(k) = |k| \mod N
  \]
  is called the **division method**. If \(N\) is a prime number, the division compression map may help spreading out the hashed values. Otherwise, there is a likelihood that patterns in the key distributions are repeated (e.g. the keys \{200,205,210,\ldots,600\} and \(N=100\).
A more sophisticated compression function is the multiply add and divide (or MAD method), with the compression function:

\[ h = \lfloor ak + b \rfloor \mod N \]

where \( N \) is a prime number, and \( a, b \) are random non-negative integers selected at the time of compression, such that \( a \mod N \neq 0 \).

With this choice of compression function the probability of collision is at most \( 1/N \).
Collision Handling Schemes

- The collision occurs when for two distinct keys $k_1, k_2$ we have $h(k_1) = h(k_2)$. This prevents simple insertion of a new item $(k,e)$ into a bucket $A[h(k)]$.

- A simple way of dealing with collisions is to make each bucket capable of storing more than one item. This requires each bucket $A[i]$ to be implemented as a sequence, list or vector. This collision resolution rule is known as **separate chaining**. Each bucket can be implemented as a miniature dictionary.

\[ (41, 28, 54, 19, 33, 21, 90, 47) \]

\[ h(k) = |k| \mod 7 \]
Load Factors and Rehashing

- If a good hashing function is used for storing $n$ items of a dictionary in a bucket of size $N$, we expect each bucket to be of size $\sim n/N$. This parameter (called load factor) should be kept small. If this requirement is fulfilled, the running time of methods findElement, insertItem and removeElement is $O(n/N)$.

- In order to keep the load factor below a constant the size of bucket array needs to be increased and to change the compression map. Then we need to re-insert all the existing hash table elements into a new bucket array using the new compression map. Such size increase and the hash table rebuild is called rehashing.
Open Addressing

- Separate chaining allows simple implementation of dictionary operations, but requires the use of an auxiliary data structure (a list, vector or sequence).
- An alternative is to store only one element per bucket. This requires a more sophisticated approach of dealing with collisions.
- Several methods for implementing this approach exist referred to as open addressing.
Linear Probing

- In this strategy, if we try to insert an item \((k,e)\) into a bucket \(A[h(k)]=A[i]\) that is already occupied, we try to insert an element in the positions \(A[(i+j)\mod N]\) for \(j=1,2,\ldots\) until a free place is found.
- This approach requires re-implementation of the method \(\text{findElement}(k)\). To perform search we need to examine consecutive buckets starting from \(A[h(k)]\) until we either find an element or an empty bucket.
- The operation \(\text{removeElement}(k)\) is more difficult to implement. The easiest way to implement it is to introduce a special “deactivated item” object.
- With this object, \(\text{findElement}(k)\) and \(\text{removeElement}(k)\) should be implemented should skip over deactivated items and continue probing until a desired item or an empty bucket is found.
Linear Probing

- The algorithm for `insertItem(k,e)` should stop at the deactivated item and replace it with `e`.
- In conclusion, linear probing saves space, but complicates removals. It also clusters of a dictionary into contiguous runs, which slows down searches.

An insertion into a hash table with linear probing. The compression map is $h(k) = k \mod 11$
Quadratic Probing

- Another open addressing strategy (known as **quadratic probing**) involves iterative attempts to buckets $A[(i + j^2) \mod N]$ for $j=0,1,2,...$ until an empty bucket is found.

- It complicates the removal operation, but avoids clustering patterns characteristic with linear probing.

- However, it creates **secondary clustering** with a set of secondary cells bouncing around a hash table in a fixed pattern.

- If $N$ is not a prime number, quadratic probing may fail to find an empty bucket, even if one exists. If the bucket array is over 50% full, this may happen even if $N$ is a prime.
Double Hashing

- Double hashing strategy eliminates the data clustering produced by the linear or quadratic probing.

- In this approach a secondary hash function $h'$ is used. If the primary function $h$ maps a key $k$ to a bucket $A[h(k)]$ that is already occupied, the the following buckets are attempted iteratively:

$$A[(i + j \cdot h'(k)) \mod N], \quad j = 1,2,...$$

- The second hash function must be non-zero. For $N$ a prime number, the common choice is

$$h'(k) = q - (k \mod q)$$

with $q < N$ a prime number.

- If memory size is not an issue, the separate chaining is always competitive to collision handling schemes based on open addressing.
Ordered dictionaries

In an ordered dictionary we use a comparator to provide the order relation among the keys. Such ordering allows efficient implementation of the dictionary ADT.

An ordered dictionary supports the following methods:

- `closestKeyBefore(k)` – returns the largest key that is $\leq k$;
- `closestElemBefore(k)` – returns $e$ with the largest key that is $\leq k$;
- `closestKeyAfter(k)` – returns the smallest key that is $\geq k$;
- `closestElemAfter(k)` – returns $e$ with the smallest key that is $\geq k$;

The ordered nature of the above operations makes the use of a log file or a hash table inappropriate for implementing the dictionary (neither of the two data structures has any ordering among the keys).
Sorted Tables

- If a directory $D$ is ordered, the items can be stored in a vector $S$ by non-decreasing order of the keys.
- The ordering of the keys allows faster searching than in the case of un-ordered sequences (possibly implemented as a linked list).
- The ordered vector implementation of a dictionary $D$ is referred to as the lookup table.
- The implementation of insertItem$(k,e)$ in a lookup table takes $O(n)$ time in the worst case, as we need to shift up all the items with keys greater than $k$ to make room for the new item.
- On the other hand, the operation findElement is much faster in a sorted lookup table than in a log file.
Binary Search

- If $S$ is an ordered sequence, then the element at the rank (position) $i$ has a key that is not smaller than the keys of the items at ranks $0,\ldots,i-1$ and no larger than the keys of the items at ranks $i+1,\ldots,n-1$.

- This ordering allows quick searching of the sequence $S$ using a variant of the game “high-low”. The algorithm has two parameters: high and low. All the candidates for a sought element at a current stage of the search are bracketed between these two parameters, i.e. they lie in the interval $[\text{low},\text{high}]$.

- The algorithm starts with the values low=0 and high=$n-1$. Then the key $k$ of the element we are searching for is compared to a key of the element at a half of $S$, i.e. $\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor$. Depending on the outcome of this comparison we have 3 possibilities:
  - If $k=\text{key(mid)}$, the item we are searching for is found and the algorithm terminates returning $e(\text{mid})$;
  - If $k<\text{key(mid)}$, then the element we are searching for is in the lower half of the vector $S$, and we set high=mid-1 and call the algorithm recursively;
  - If $k>\text{key(mid)}$, the element we are searching for is in the upper half of the vector $S$, and we set low=mid+1 and call the algorithm recursively;
Binary Search

- Operation findElement($k$) on an $n$-item ordered dictionary implemented with a vector $S$ reduces to calling BinarySearch($S,k,0,n-1$).

**Algorithm** BinarySearch($S,k$,low,high):

- **Input:** An ordered vector $S$ storing $n$ items, a search key $k$, and the integers low and high;
- **Output:** An element of $S$ with the key $k$, otherwise an exception;

  if low>high then
    return NO_SUCH_KEY
  else
    mid= $\lfloor (\text{low} + \text{high})/2 \rfloor$
    if $k$=key(mid) then
      return e(mid)
    elseif $k$<key(mid) then
      BinarySearch($S,k$,low,mid-1)
    else
      BinarySearch($S,k$,mid+1,high)
  endif
end if
Binary Search

An example of binary search to perform the operation findElement(22)
Binary Search

- Considering the computational cost of binary search, we need to notice first that at each call of the algorithm there is a constant number of operations. Thus the running time is proportional to the number of recursive calls.

- At each recursive call the number of candidates that need to be searched is high-low+1, and it is reduced by at least a half at each recursive call.

- If $T(n)$ is the computational cost of binary search, then:

\[
T(n) = \begin{cases} 
  c & \text{if } n < 2 \\
  T(n/2) + c & \text{otherwise}
\end{cases}
\]

- In the worst case the search stops when there are no more candidate items. Thus, the maximal number of recursive calls is $m$, such that

\[ \frac{n}{2^m} < 1 \]

- This implies that $m = \lceil \log n \rceil + 1$, i.e. BinarySearch($S,k,0,n-1$) runs in $O(\log n)$ time.
Binary Search Trees

- It is a tree data structure adapted to a binary search algorithm.
- A binary search tree is a binary tree in which each node stores an element $e$ and that the elements in the left subtree of that node are smaller or equal to $e$, while the elements in the right subtree of that node are greater or equal to $e$.
- An inorder traversal of a binary search tree visits the elements in a non-decreasing order.
- A binary search tree can be used to search for an element by traversing down the tree. At each node we compare the value we are searching for $x$ with $e$. There are 3 outcomes:
  - If $x=e$, the search terminates successfully;
  - If $x<e$, the search continues in the left subtree;
  - If $x>e$, the search continues in the right subtree;
  - If the whole subtree is visited and the element is not found, the search terminates unsuccessfully;
Binary Search Trees

Searching for the element 36
Searching for the element 36

Binary Search Trees

36 < 58
Binary Search Trees

Searching for the element 36

36 > 31
Searching for the element 36
Searching for the element 36

36 = 36 - success
Computational Cost of Binary Tree Searching

- The binary tree search algorithm executes a constant number of operations for each node during the traversal.
- The binary search algorithm starts from the root and goes down one level at the time.
- The number of levels in a binary search tree is called the height $h$.
- The method findElement runs in $O(h)$ time. This can potentially be a problem as $h$ can potentially be close to $n$. Thus, it is essential to keep the tree height optimal (as close to $O(\log n)$ as possible). The way to achieve this is to balance a tree after each insertion (AVL trees).
Dictionary Search Using a Binary Search Tree

The method findElement\( (k) \) can be performed on a dictionary \( D \) if we store \( D \) as a binary search tree and call the method TreeSearch\( (k, T.\text{root}()) \).

**Algorithm** TreeSearch\( (k, v) \);

**Input:** A search key \( k \) and a node \( v \) of a binary search tree;

**Output:** A node \( w \) of \( T \) equal to \( k \), or an exception;

- if \( k = \text{key}(v) \) then return \( v \);
- else if \( k \) is an external node then return NO_SUCH_KEY;
- else if \( k < \text{key}(v) \) then return TreeSearch\( (k, T.\text{leftChild}(v)) \);
- else return TreeSearch\( (k, T.\text{rightChild}(v)) \);

end if
Insertion into a Binary Search Tree

To perform the operation `insertElem(k,e)` into a dictionary $D$ implemented as a binary search tree, we call the method `TreeSearch(k,T.root())`. Suppose that $w$ is the node returned by `TreeSearch`. Then:

- If besides $w$ a flag NO_SUCH_KEY is returned, then compare $e$ with $w$. If $e < w$, create a new left child and insert the element $e$ with the key $k$. Otherwise, create a right child and insert the element $e$ with the key $k$.
- If only the node $w$ is returned (there is another item with key $k$), we call the algorithm `TreeSearch(k,T.leftChild(w))` and `TreeSearch(k,T.rightChild(w))` and recursively apply the algorithm returned by the node `TreeSearch`. 
Insertion into a Binary Search Tree

Insertion of an item with the key 78 into a binary search tree
Insertion into a Binary Search Tree
Removal from a Binary Search Tree

- Performing removeElement\((k)\) on a dictionary \(D\) implemented with a binary search tree introduces an additional difficulty that the tree needs to remain connected after the removal.

- We need to execute first TreeSearch\((k, T.\text{root}())\) to find a node with a key \(k\). If the algorithm returns an exception, there is no such element in \(D\). If the key \(k\) is found in \(D\), we distinguish two cases:
  - If the node with the key \(k\) is the leaf node, the removal operation is simple;
  - If the node with the key \(k\) is an internal node, its simple removal would create a hole. To avoid this, we need to do the following:
Removal from a Binary Search Tree

1. Find the first node $y$ that follows $w$ in an inorder traversal. It is the leftmost internal node in the right subtree of $w$ (go right from $w$, and then follow the left children;)
2. Save the element stored at $w$ into a temporary variable $t$, and move $y$ into $w$. This would remove the previously stored element of $w$;
3. Remove the element $y$ from $T$;
4. Return the element stored in a temporary variable $t$;
Removal from a Binary Search Tree

Removal of the element with the key 65
Removal of the element with the key 65
Removal from a Binary Search Tree

Removal of the element with the key 65
AVL Trees

- The idea behind introducing the AVL trees is to improve the efficiency of the basic operations for a dictionary.
- The main problem is that if the height of a tree that implements a dictionary is close to $n$, the basic operations execute in time that is asymptotically no better than that obtained from the dictionary implementations via log files or lookup tables.
- A simple correction is to have an additional property added to the definition of a binary search tree to keep the logarithmic height of the tree. This is the height balance property:
  
  For every internal node $v$ of the tree $T$, the heights of its children can differ by at most 1.
AVL Trees

- Any subtree of an AVL tree is an AVL tree itself.
- The height of an AVL tree that stores $n$ items is $O(\log n)$.
- This implies that searching for an element in a dictionary implemented as an AVL tree runs in $O(\log n)$ time.

An example of an AVL tree with the node heights
The first phase of an element insertion into an AVL tree is the same as for any binary tree.

An example of inserting the element with the key 54 into an AVL tree
The first phase of an element insertion into an AVL tree is the same as for any binary tree.

An example of inserting the element with the key 54 into an AVL tree
Insertion into an AVL Tree

- The first phase of an element insertion into an AVL tree is the same as for any binary tree.

An example of inserting the element with the key 54 into an AVL tree.
Insertion into an AVL Tree

- Suppose that the tree satisfies the height-balance property prior to the insertion of a new element \( w \). After inserting the node \( w \), the heights of all nodes that are on the path from the root to the newly inserted node will increase. Consequently, these are the only nodes that may become unbalanced by the insertion.

- We restore the balance of the nodes in the AVL tree by a search and repair strategy.

- Let \( z \) be the first node on the path from \( w \) to root that is unbalanced.

- Denote by \( y \) the child of \( z \) with a larger height (if there is a tie, choose \( y \) to be an ancestor of \( w \)).

- Denote by \( x \) the child of \( y \) with a larger height (if there is a tie, choose \( x \) to be an ancestor of \( w \)).
Insertion into an AVL Tree
Balancing an AVL Tree

- The node $z$ becomes unbalanced because of an insertion into the subtree rooted at its child $y$.
- The subtree rooted at $z$ is rebalanced by the trinode restructuring method. There are 4 cases of the restructuring algorithm. The modification of a tree $T$ by a trinode restructuring operation is called a rotation. The rotation can be single or double.
- The trinode restructuring methods modify parent-child relationships in $O(1)$ time, while preserving the inorder traversal ordering of all nodes in $T$. 
Trinode Restructuring by Single Rotation

1. Original structure:
   - Node Z is connected to Y, which is connected to X.

2. Single rotation:
   - Rotate around Y to change the structure.

3. New structure:
   - Node Z is connected to Y, which is connected to X.

4. Repeat:
   - Another single rotation around Y.

5. Final structure:
   - Node Z is connected to Y, which is connected to X.
Trinode Restructuring by Double Rotation

double rotation

double rotation
Balancing an AVL Tree
Removal from an AVL Tree

- The first phase of the element removal from an AVL tree is the same as for a regular binary search tree. This can, however, violate the height-balance property of an AVL tree.

- If we remove an external node, the height balance property will be satisfied.

- But, if we remove an internal node and elevating one of its children into its place, an unbalanced node in $T$ may occur. This node will be on the path from the parent $w$ of the previously removed node to the root of $T$.

- We use the trinode restructuring after the removal to restore the balance.
Removal from an AVL Tree

- Let $z$ be the first node encountered going upwards from $w$ (the parent of the removed node) towards the root of $T$.
- Let $y$ be the child of $z$ with a larger height (i.e. it is a child of $z$, but not an ancestor of $w$).
- Let $x$ be a child of $y$ with a larger height (this choice may not be unique).
- The restructuring operation is then performed locally, by restructuring a subtree rooted in $z$. This may not recover the height balance property, so we need to continue marching up the tree and looking for the nodes with no height balance property.
- The operation complexity of the restructuring is proportional to the height of a tree, hence $O(\log n)$.
Removal of the element with key 32 from the AVL tree
Removal from an AVL Tree