CS C341 / IS C361

Data Structures & Algorithms

3/15/2014

Dictionary Data Structures - Hashing

Open Addressing

- Analysis
- Probing
- Unsuccessful Find
- Successful Find

Bloom Filters

- Motivation
- Implementation
- Analysis
- Applications.

TERMINOLOGY

- The technique of chaining elements that hash into the same slot is referred to by different names:
  - Separate Chaining
    - for obvious reasons
  - Open Hashing
    - because number of elements is not limited by table size
  - Closed Address Hashing
    - because the location of the bucket (i.e. the address) of an element is fixed
      - Consider an element e that is added, removed, and added again:
        - it will get added to the same bucket.

Open Addressing (a.k.a. closed hashing)

- Fixed Space
  - Fixed Table size and
  - each table location can contain only one element
- Addressing by Hashing
  - Same as in Separate Chaining
- Probing (for a vacant location) in case of collision

OPEN ADDRESSING (A.K.A. CLOSED HASHING)

- add(Element e, Hashtable T)   // Generic procedure

```
// e.key is key; h is hash function
a = h(e.k);
if T[a] is empty then { T[a]=e; return; }
j=0;
repeat  {
   j++;
  b = getNextAddr(a,k,j);
} until (T[b] is empty)        // Will this terminate?
T[b] = e;
```

Open Addressing – Probing Schemes

// m denotes table size; typically m is chosen to be prime

- Linear Probing:

  getNextAddr(a,k, j) { return (a+j) mod m; }

- Quadratic Probing

  getNextAddr(a,k,j) { return $(a+j^2)$ mod m; }

- Exponential Probing

  getNextAddr(a,k,j) { return $(a+2^j)$ mod m; }

- Double Hashing

  getNextAddr(a,k,j) { return a+j*h_2(k) mod m; }

  // $h_2(k)$ is the secondary hash function

  // $h_2(k)$ must be non-zero

  // e.g. $h_2(k) = q - (k \bmod q)$ for some prime $q < m$

3/15/2014

Open Addressing

- Implementation Caveat:
  - add as defined may not terminate!
    - Must check whether all ***m*** locations have been probed
      - Could be expensive!
    - Alternatively, may use a count of non-empty locations.
      - Will work only if the probing sequence covers all locations

Exercise: Handle termination: use simple heuristics(s). **End of Exercise.**

Open Addressing

- Define find.
  - Similar to add:
    - hash
    - if element found return it;
    - if empty return INVALID;
    - otherwise probe until element found or empty slot.
      - return accordingly.
  - Termination?

Open Addressing

- How is deletion done?
  - Deleted slots must be marked *deleted*
    - ***deleted*** flag different from ***empty*** flag for probing procedure to work
      - **find** will treat ***deleted*** slots as ***empty*** slots
  - This won't allow re-use of ***deleted*** slots
    - How do you recover deleted slots?
      - **add** can be modified to fill in any *deleted* slot encountered in a probing sequence
        - This may not cover all deleted slots
      - **delete** can be implemented such that subsequent entries in a probing sequence are pulled in.
- How does your deletion scheme affect further probes?

Open Addressing – Analysis of Probing

- Probing sequence:
  - Sequence of slots generated: S[k,0], S[k,1],..S[k,m]
- Probing requirements:
  - Utilization:
    - The probing sequence must be a permutation of 0,1,…m-1
  - Uniform hashing assumption:
    - Requires that each key may result in any of the m! probing sequences

Open Addressing – Analysis of Probing [2]

- Linear Probing
  - Slot for the j^{th} probe in a table of size m

    S[k,j] = (h(k) + j) mod m
  - Long runs of occupied slots build up
    - If an empty slot is preceded by j full slots,
    - then the probability this slot is the next one filled is (j+1)/m
    - instead of 1/m (in a table of size m)
  - Effect known as (Primary) clustering
- This is not a good approximation of uniform hashing

Open Addressing – Analysis of Probing [3]

- Quadratic Probing
  - Slot for the j^{th} probe in a table of size m

    $S[k,j] = (h(k) + j^2) \bmod m$
  - Clustering effect milder than Linear probing
    - Effect known as secondary clustering
  - But the sequence of slots probed is still dependent on the initial slot (decided by the key)
    - i.e only m distinct sequences are explored
- Generalize:
    - $S[k,j] = (h(k) + a*j + b*j^2) \bmod m$

Exercise: Can you choose a, b, and m such that all slots are utilized?

Exercise: Repeat (very similar) analysis for Exponential Probing.

OPEN ADDRESSING – ANALYSIS OF PROBING [4]

- Double Hashing
  - Slot for the j^{th} probe in a table of size m

    $S[k,j] = (h_1(k) + j*h_2(k)) \bmod m$
  - Probing sequence depends on k in two ways
    - So, probing sequence depends not only on initial slot

      i.e. m*m probing sequences can be used.

    This results in behavior closer to uniform hashing
  - If $\gcd(h_2(k),m) = d$ for some key k,
    - then the sequence will explore only $(1/d)*m$ slots
      - Why?
    - So, choose (for instance):
      - m as a prime, and ensure $h_2(k)$ is always < m
- Can you extend this to a sequence of hashes $h_1(k)$, $h_2(k)$, $h_3(k)$, … ?

Open Addressing – Analysis - Unsuccessful Find

- Given: open-address table with load factor $\alpha = n/m < 1$
- Assumption: Uniform Hashing
- Expected number of probes in an unsuccessful find is at most

  $1/(1-\alpha)$
- Proof:
  - Last probed slot is empty; all previous probed slots are non-empty but do not contain the given key
  - Define p_j as the probability that exactly j probes access non-empty slots
    - Then the expected number of probes is $1 + \sum_0^\infty j*p_j$
  - If q is defined as the probability that at least j probes access non-empty slots then $\sum_0^\infty j*p_j = \sum_1^\infty q_j$

Open Addressing – Unsuccessful Find

- Proof: (contd.)
  - The expected number of probes is

    $1 + \sum_0^\infty j*p_j = 1 + \sum_1^\infty q_j$
  - With uniform hashing

    $q_j = (n/m) * ((n-1)/(m-1)) * \ldots ((n-j+1)/(m-j+1))$

    $<= (n/m)^j$
  - Then the expected number of probes is

    $1 + \sum_1^\infty q_j <= 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$

    $= 1/(1 - \alpha)$

Open Addressing – Analysis - successful Find

- Given: open-address table with load factor $\alpha = n/m < 1$
- Assumptions: Uniform Hashing; All keys are equally likely to be searched
- Expected number of probes in a successful find is at most $1/\alpha + (1/\alpha)*\ln(1/(1-\alpha))$
- Proof:
  - Simplifying assumption :
    - A successful find follows the same probe sequence as when the element was inserted
    - When is the assumption reasonable?
  - If k was the $(j+1)^{st}$ key to be inserted
    - then the expected number of probes in finding k is given by the previous theorem (on unsuccessful find)

$$1/(1 - (j/m)) = m/(m-j)$$

Open Addressing – analysis - successful Find

- Proof: (contd.)
  - Expected number of probes in finding the key that was inserted as the $(j+1)^{st}$ is $m/(m-j)$
  - Average over all n keys in the table

$$(1/n) \sum_{j=0}^{n-1} (m/(m-j)) = (m/n)* (\sum_{j=0}^{n-1} (1/(m-j)))$$
$$= (1/\alpha) * (H_m - H_{m-n})$$

where H_m is the m^{th} Harmonic number.

  - Since $\ln(j) <= H_j <= 1+ \ln(j)$

$$(1/\alpha) * (H_m - H_{m-n}) <= (1/\alpha) * (1 + \ln m - \ln(m-n))$$
$$= (1/\alpha) + (1/\alpha) * \ln (m/m-n)$$
$$= 1/\alpha + (1/\alpha)*\ln(1/(1-\alpha))$$

Re-Hashing

- Hash tables support efficient find operations:
  - Average case time complexity is O(1) if load factor is low
    - Load factor must be < 1 for separate chaining
- In practice,
  - Load factor must be < 0.75 to expect good performance.
- What if the hash table is nearly "full"?
  - Extend the hash table (i.e. increase its size)
    - Can the new hash function assign the old values to the same buckets as before?
      - bucket addresses must change for a good distribution?
  - Re-insert all the elements in the table
    - Referred to as ***re-hashing.***

Re-Hashing

- Cost of Rehashing
  - O(max(m,n)) time – typically O(n) as table is nearly full.
    - Amortized Cost: O(1) time per element
  - But response time at the point of rehashing is bad:
    - allocation and copying of all the values takes O(n) time between two operations.
    - Or between the request for an operation and the response.
  - This is bad for applications requiring
    - bounded (worst case) response time
- What should be the size of the extended table?
  - Typical choice: 2*|T|
  - Trade-offs: ???

Bloom Filters - Motivation

- Tradeoff: Space vs. (In)Correctness
  - i.e. storage space for the table vs. false positives (membership)
- Example Problem: Stemming of words in search engine indexing:
  - e..g. plurals stemmed to singular; all parts of speech stemmed to one form
    - 90% of cases can be handled by simple rules
      - Rest - the exceptions – need a dictionary lookup
    - Suppose dictionary is large and must be stored in disk

BLOOM FILTERS - MOTIVATION

- Consider this outline for stemming :

  for each word w

  if (w is an exception word)

  then getStem(w,D)

  else apply-simple-rule(w)

  Need dictionary lookup on disk

- Cost for checking exceptions:
  - $N * T_d$ where
    - N is # words and
    - T_d is lookup time (on disk)

Bloom Filters - Motivation

- Suppose we can trade-off space for false positives (in lookup):

  for each word w

      if (w is in Dm)   // in-memory lookup (probabilistic)

      then { s = getStem(w, Dd);   // disk lookup (deterministic)

        if invalid(s) then apply-simple-rule(w);

      } else { apply-simple-rule(w); }

- Cost for checking exceptions:
  - $N * T_m + (r + f)*N*T_d$
    - r is the proportion of exception words
    - f is false positive rate
    - T_m is lookup time in memory
    - T_d is lookup time on disk
  - Time Saved: $(1 - r - f) * (T_d - T_m) / T_d$

Bloom Filters – An implementation

- Hash table is an array of bits indexed from 0 to m-1.
  - Initialize all bits to 0.
  - insert(k):
    - Compute $h_1(k)$, $h_2(k)$, …, $h_d(k)$ where each h_i is a hash function resulting in one of the m addresses.
    - Set all those addressed locations to 1.
  - find(k):
    - Compute $h_1(k)$, $h_2(k)$, …, $h_d(k)$
    - If all addressed locations are 1 then k is **found**

      Else k is **not found**

Not necessarily correct!

Always correct.

Bloom Filters - analysis

- Consider a table H of size m.
- Assume we use d "good" hash functions.
- After n elements have been inserted, the probability that *a specific location is 0* is given by
  - $p = (1 – 1/m)^{dn} \approx e^{-dn/m}$
- Let q be the proportion of 0 bits after insertion of n elements
  - Then the expected value E(q) = p
- Claim (w/o proof):
  - With high probability q is close to its mean.
- So, the false positive rate is:
  - $f = (1-q)^d = (1-p)^d = (1 - e^{-dn/m})^d$

Bloom Filters

- The data structure is probabilistic:
  - If a value is not found then it is definitely not a member
  - If a value is found then it may or may not be a member.
- The error probability can be traded for space.
  - In practice, one can get low error probability with a (small) constant number of bits per element: (1 in our example implementation) .
- Applications:
  - Dictionaries (for spell-checkers, passwords, etc.)
  - Distributed Databases – exchange Bloom Filters instead of full lists.
  - Network Processing – Caches – exchange Bloom Filters instead of cache contents
  - Distributed Systems – P2P hash tables : instead of keeping track of all objects in other nodes, keep a Bloom filter for each node.

Las Vegas vs. Monte Carlo

- Quicksort:
  - Randomization for improved performance – correctness not altered
- Hashtables (for unordered dictionaries) :
  - Any 1-to-1 mapping will yield a table but a good hash function should yield a “uniformly random” distribution
  - Universal hashing chooses hash function “randomly”
- Both of the above are optimizations:
  - Such techniques are referred to as Las Vegas techniques.
- Monte Carlo Technique
  - Bloom Filter - Randomization yields a probabilistic algorithm that does not always produce correct results.

CS C341 / IS C361
Data Structures & Algorithms

Dictionary Data Structures – Search Trees

Comparison of Sorted Arrays and Hashtables

Ordered Dictionaries

- Better Representation
- Binary Trees
- Binary Search Trees
    - Implementation (Find, Add, and Delete)
    - Efficiency
    - Order Queries

Balancing a Search Tree

- Height Balance Property

DICTIONARY IMPLEMENTATIONS - COMPARISON

Sorted Array

- Suitable for:
  - Ordered Dictionary
    - Example Queries: 2nd largest element? OR the element closest to k?
  - Offline operations (insertions/deletions)
  - Comparable Keys
- Implementation:
  - Deterministic

Hashtable

- Suitable for:
  - Unordered Dictionary
  - Online insertions (deletions??)
    - Resizing can be done at an amortized cost of O(1) per element
  - Hashable Keys
- Implementation:
  - Randomized

DICTIONARY IMPLEMENTATIONS - COMPARISON

Sorted Array

- Time Complexity (find):
  - Θ(logN) - worst case and average case
- Space Complexity
  - Θ(1)

Hashtable

- Time Complexity (find):
  - Θ(1) average case and Θ(N) worst case
- Space Complexity
  - Θ(N) words – separate chaining (links)
  - Θ(N) bits – open addressing (empty & deleted flags)

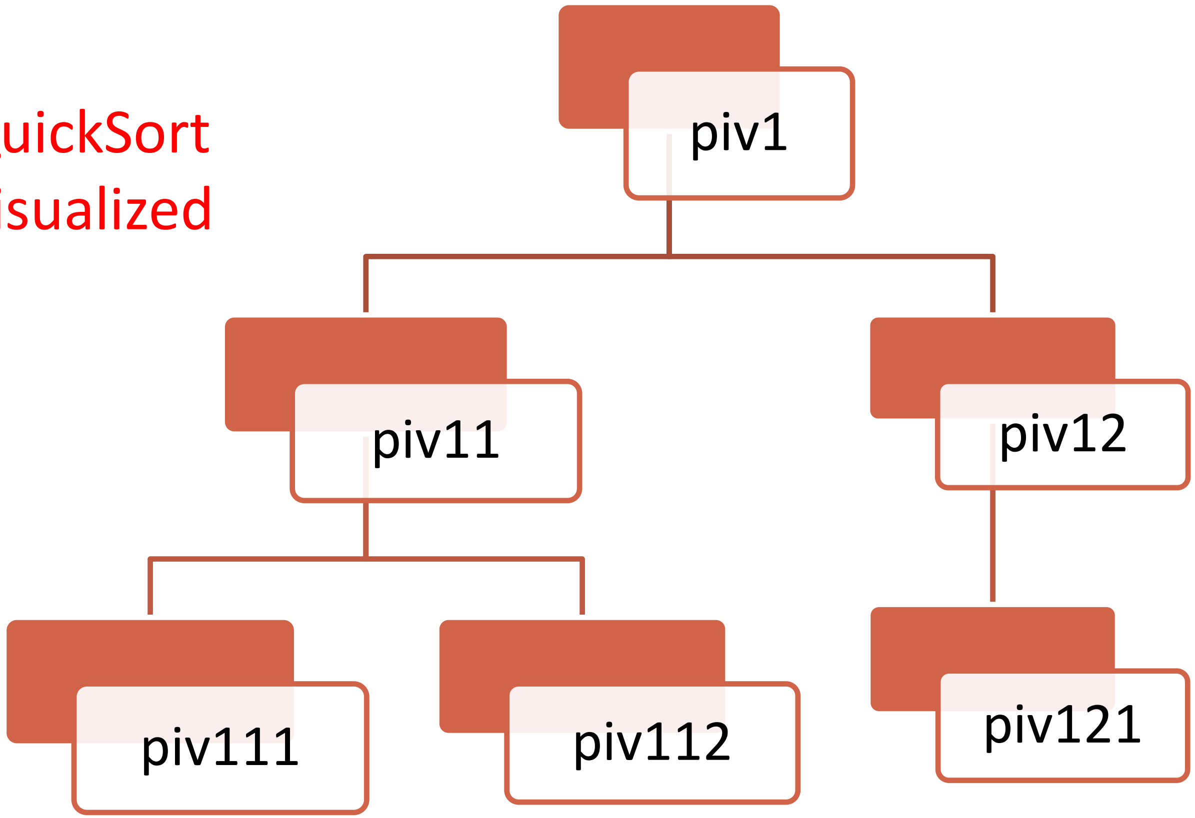
Ordered Dictionary – Better Representation?

- Is there an representation that
  - supports "relative order" queries and
  - supports online operations and
  - is resizable ?
- Revisit (the general structure of) Quicksort(Ls)

```
Quicksort(Ls) {
 If (|Ls|>0) {
   Partition Ls based on a pivot into LL and LG
   QuickSort LL
   QuickSort LG
 }
}
```

ORDERED DICTIONARY – BETTER REPRESENTATION?

QuickSort Visualized



Ordered Dictionary – Better Representation?

- Can we re-materialize the *QuickSort order* while searching?
  - i.e. a representation where *key* is compared with the *pivot* (pre-selected)
    - key == pivot ==> done
    - key < pivot ==> search in left subset
    - key > pivot ==> search in right subset.
- This is similar to QuickSelect but
  - With pre-selected pivots and stored "ordering" between the pivots.
    - i.e. ordering is preserved after sorting so as to support to "relative order" queries

Ordered Dictionary – Better Representation?

- Data Model:
  - A Set is characterized by the “Relation between Pivot and two (sub)sets”
- Generalized Data Model:
  - A set is characterized by a “root” element and two subsets.

Ordered Dictionary – Better Representation?

- Inductive Definition:
  - A binary tree is
    1. empty OR
    2. made of a root element and two binary trees - referred to as left and right (sub) trees
  - For induction to be well founded "sub trees" must be of smaller size than the original.
  - Sub trees are referred to as children (of the node which is referred to as the parent)
  - A binary tree with two empty children is referred to as a leaf.
- Inductive Definitions can be captured recursively:

BinaryTree = EmptyTree U (Element x BinaryTree x BinaryTree)

ADT Binary Tree

- BinaryTree createBinTree() // create empty tree
- Element getRoot(BinaryTree bt)
- BinaryTree getLeft(BinaryTree bt)
- BinaryTree getRight(BinaryTree bt)
- BinaryTree compose(Element root,
  BinaryTree leftBt,
  BinaryTree rightBt)

ADT Binary Tree - Representation

- struct __binTree;
- typedef struct __binTree *BinaryTree;
- struct __binTree { Element rootVal;

  BinaryTree left;

  BinaryTree right;

  };

Argue that the above representation in C captures the definition:

BinaryTree = EmptyTree U (Element x BinaryTree x BinaryTree)

ADT Binary Tree - Implementation


```
BinaryTree compose(Element e, BinaryTree lt, BinaryTree rt)
{
   BinaryTree newT =
          (BinaryTree)malloc(sizeof(struct   __binTree));
    newT->rootVal = e;
    newT->left = lt;
    newT->right = rt;
    return newT;
 }
```

Ordered Dictionary – Search Tree

- A binary search tree is
- a binary tree that captures an "ordering" (i.e. a relation) $\leq$ via the relation between the root and its subtrees:
  - i.e. for each element *eL* in the left subtree:
    - *eL* $\leq$ *rootVal*
  - and for each element *eR* in the right subtree:
    - *rootVal* $\leq$ *eR*

ADT Ordered Dictionary

- Element find(OrdDict d, Key k)
- OrdDict insert(OrdDict d, Element e)
- OrdDict delete(OrdDict d, Key k)
  - Note on Representation:
    - We can use the same BinaryTree representation for this.
      - i.e. The ordering is captured implicitly at the point of insertion by leveraging the left and right information.
    - Hence the following type definition – in C – would serve as the data definition!
  - End of Note.
- typedef BinaryTree OrdDict;

ADT Ordered Dictionary – Implementation

```
//Preconditions:   k is unique;
Element  find(OrdDict d, Key k)
{
   if (d==NULL) return NOT_FOUND;
   if (d->rootVal.key == k) return d->rootVal;
    else if (d->rootVal.key < k) return find(d->right, k);
    else /* d->rootVal.key > k */  return find(d->left, k);
}
```

(Trivial) Exercise: Modify implementation for multiple elements with the same key value.

End of Exercise.

ADT Ordered Dictionary - Implementation


```
//Preconditions:  d is non-empty;  keys are unique (i.e. duplicates);
OrdDict  insert(OrdDict d, Element e)
{
  if (d->rootVal.key < e.key)  {
    if (d->right == NULL) { d->right = makeSingleNode(e); }
    else {  insert(d->right, e);  }
   } else  {
    if (d->left == NULL) {  d->left = makeSingleNode(e); }
    else {  insert(d->left, e);  }
   }
   return d;
} /* Exercise: Modify the top-level procedure to handle the case of the
    "empty tree".
Modify the procedure to handle duplicates.
End of Exercise. */
```

ADT Ordered Dictionary - Implementation

```
void  makeSingleNode(Element e)
{
     OrdDict node;
     node = (OrdDict) malloc(sizeof(struct  __binTree));
    node->rootVal=e;
    node->left = node->right = NULL;
     return node;
}
```

Exercise: Modify implementation for multiple elements with the same key (use one of the options):

- return success but do nothing,
- return failure with message “already found”,
- return success after adding new element separately,
- return success after overwriting contents.

End of Exercise

ADT Ordered Dictionary - Implementation

OrdDict delete(OrdDict dct, Key k)

- find the node, say nd, with contents matching key k
- if no such node exists done

  else if nd is a leaf then delete nd // must free nd

  else if one of the children of nd is empty

  then replace nd with the other subtree of nd

  else

  in-order successor of nd will : (i) be within the subtree and (ii) have an empty left subtree

  a. find in-order successor of nd, say suc
  b. swap contents of suc with nd
  c. if suc is a leaf-node then delete suc // must free suc

     else replace suc with its right sub-tree

ADT Ordered Dictionary - Implementation

```
OrdDict  delete(OrdDict dct, Key k)
{
   if (dct==NULL) return NULL;
   for (par=NULL, nd=dct; nd!=NULL; ) {
     if (nd->rootVal.key==k)  break;
     else if (nd->rootVal.key < k) { par=nd; nd=nd->right;}
     else { par=nd; nd=nd->left;  }
    }
    if (nd==NULL) return dct;
    if (par==NULL) {  free(nd); return NULL; }
    else { return deleteSub(par, nd); }
}
```

ADT Ordered Dictionary - Implementation

```
OrdDict  deleteSub(OrdDict  par,  OrdDict toDel) {
     if (toDel->left!=NULL && toDel->right!=NULL) {
          return deleteSubReplace(par, toDel);
    } else if (toDel->right!=NULL) {
          if (par->left==toDel) { par->left=toDel->right; }
          else { par->right=toDel->right; }
    } else if (toDel->left!=NULL) {
          if (par->left==toDel) { par->left=toDel->left; }
          else { par->right=toDel->left; }
    } else {
          if (par->left==toDel) {par->left=NULL;}
            else {par->right=NULL;}
     }
    free(toDel); return dct;
}
```

find in-order successor of nd, say suc

a. swap contents of suc with nd
b. if suc is a leaf-node then delete suc // must free suc

else replace suc with its right sub-tree

ADT Ordered Dictionary - Implementation

```
OrdDict  deleteSubReplace(OrdDict  par,  OrdDict del)
{
     for (par=del,suc=del->right; suc->left!=NULL; par=suc,suc=suc->left) ;
     swapContents(del, suc);
     if (suc->right==NULL) {
          if (par->left==suc) {par->left=NULL;}
          else {par->right=NULL; }
    } else {
          if (par->left==suc) { par->left=suc->right; }
          else { par->right=suc->right; }
    }
    free(suc); return dct;
}
```

ADT Ordered Dictionary - Complexity

- Time Complexity:
  - Find, insert, delete
    - Height of the tree
- Height of binary tree (by induction):
  - Empty Tree ==> 0
  - Non-empty ==> 1 + max(height(left), height(right))
- Balanced Tree
  - Height = logN
    - Why?
- Unbalanced Tree
  - Worst case height = N
    - Example?

Binary Search Trees (BSTs)

- BSTs store data in order:
  - i.e. if you traverse a BST such that for all nodes v,
    - Visit all nodes in the left sub tree of v
    - Visit v
    - Visit all nodes in the right sub tree of v
  - then you are visiting them in sorted order.
- This is referred to as in-order traversal:

```
inorder(BinaryTree bt) {
   if (bt != NULL) {
         inorder(bt->left));
         visit(bt);
         inorder(bt->right);
   }
 }        // Time Complexity??  Space Complexity??
```

Binary Search Trees (BSTs)

- Revisiting *delete* (in an Ordered Dictionary):
  - Deletion of an element with two non-empty subtrees required a pull-up operation.
  - One way of pulling-up –
    - find an element, say c, closest to the element to be deleted, say d
      - How?
    - overwrite d with c
    - delete node (originally) containing c
      - Will this result in recursive pulling-up? Why or why not?

BINARY SEARCH TREES (BSTS)

- Revisiting *delete* (in an Ordered Dictionary):
  - Here is the ***pullUpLeft*** procedure

```
pullUpLeft(OrdDict  toDel, OrdDict cur) {
  pre = toDel;
  while (cur->right != NULL) { pre=cur; cur=cur->right; }
  toDel->rootVal = cur->rootVal;
  if (cur->left==NULL) { prev->right = NULL; }
  else { prev->right = cur->left; }
  free(cur);
}
   // Exercise: Write a pullUpRight procedure
```

Binary Search Trees – Order Queries

- Exercises:
  - Write a procedure to find the minimum element in a BST.
  - Write a procedure to find the maximum element in a BST
  - Write a procedure to find the second smallest element in a BST.
  - Write a procedure to find the k^{th} smallest element in a BST.
  - Write a procedure to find the element closest to a given element in a BST.
- Hint:
  - In all the above cases, use in-order traversal and stop once you get the result.

Binary Search Tree - Complexity

- Time Complexity:
  - Find, insert, delete
    - # steps = Height of the tree
- Height of binary tree (by induction):
  - Empty Tree ==> 0
  - Non-empty ==> 1 + max(height(left), height(right))
- Balanced Tree – Best case
  - Height = log(N) where N is the number of nodes
- Unbalanced Tree – Worst case
  - Worst case height = N   where N is the number of nodes
- How do you ensure balance?

HEIGHT-BALANCE PROPERTY

- A node v in a binary tree is said to be ***height-balanced*** if
  - the difference between the heights of the children of v – its sub-trees – is at most 1.
- Height Balance Property:
  - A binary tree is said to be ***height-balanced*** if each of its nodes is height-balanced.
- Adel’son-Vel’skii and Landis tree (or AVL tree)
  - Any height-balanced binary tree is referred to as an AVL tree.
- The height-balance property keeps the height minimal
  - How?

AVL TREE - HEIGHT

- Theorem:
  - The minimum number of nodes n(h) of an AVL tree of height h is $\Omega(c^h)$ for some constant c>1.
- Proof (By induction):
  1. n(1) = 1 and n(2) = 2
  2. For h>2, n(h) >= n(h-1) + n(h-2) + 1

     Why?
  3. Then, n(h) is a monotonic sequence i.e. n(h) > n(h-1). So, n(h) > 2*n(h-2)
  4. By, repeated substitution, $n(h) > 2^j * n(h-2*j)$ for h-2*j >=1
  5. So, n(h) is $\Omega(2^h)$

AVL Tree - Height

- Corollary:
  - The height of an AVL tree with n nodes is O(log n).
  - Proof:
    - Obvious from the previous theorem.

- Thus the cost of a ***find*** operation in an AVL tree with n nodes is O(log n).
- What about insertion and deletion?
  - Adding or removing a node may disturb the balance.