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
  }
}
```

• p p
i i
v v
1 1
2 2
2

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
  - empty OR
  - 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) **S** via the relation between the root and its subtrees:
  - i.e. for each element *eL* in the left subtree:
  - *eL* **S** *rootVal*
  - and for each element *eR* in the right subtree:
  - *rootVal* **S** *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 Exerci

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
  find in-order successor of nd, say suc
  swap contents of suc with nd
  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

- swap contents of suc with nd
- 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 kth 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 Ω(ch) 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
   3. Why?
4. Then, n(h) is a monotonic sequence i.e. n(h) > n(h-1). So, n(h) > 2*n(h-2)
5. By, repeated substitution, n(h) > 2j * n(h-2*j) for h-2*j >=1
6. So, n(h) is Ω(2h)

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.

CS C341 / IS C361
Data Structures & Algorithms

Dictionary Data Structures – Search Trees

Balancing a Search Tree
- Height Balance Property
- AVL Tree
    - Example
    - Rotations
    - Time Complexity
        - Number of rotations for insert and delete
    - Implementation issues

Binary Search Tree - Complexity

RECALL

- Time Complexity:
  - Find, insert, delete
  - # steps = Height of the tree
- 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

- RECALL
- 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.
- Theorem:
  - The minimum number of nodes n(h) of an AVL tree of height h is $\Omega(c^h)$ for some constant c>1.
- Corollary:
  - The height of an AVL tree with n nodes is O(log n).

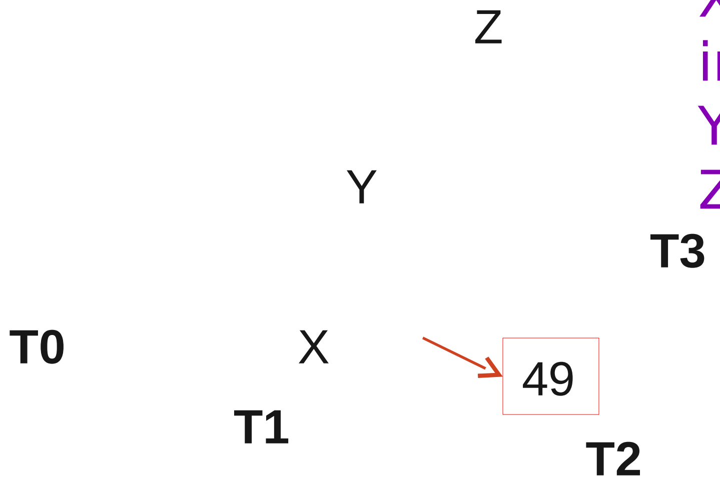
[illegible]

AVL Tree – Insertion - Example

Insert
49

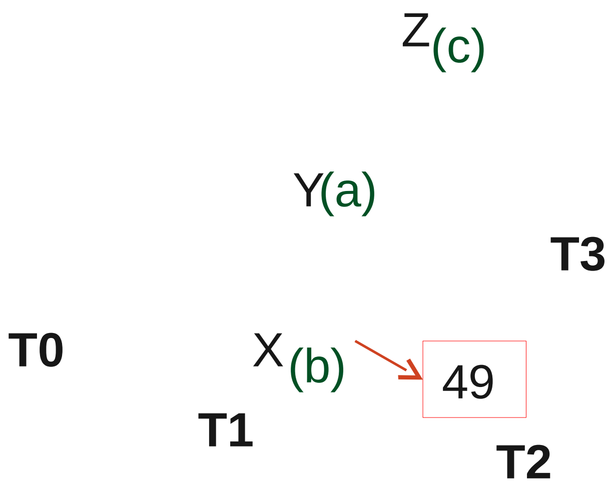
[illegible]

AVL Tree – Insertion – Example 1



X – point of insertion
Y – parent of X
Z – parent of Y

T0 – T1 : left to right listing of other sub-trees involved



(a,b,c) left-to-right listing of (X,Y,Z)

3 • 2 4 3 3
3 5 4 9 6

AVL Tree – Insertion ex.1

Re-structure:
Input: Z, a , b, c, and T1, T2, T3, T4

b

1. **Replace subtree at Z with subtree at b**

b

a

T0 T1

2. Set a as left subtree of b and set T0 and T1 as left & right subtrees of a

[illegible]

AVL Tree – Insertion – Ex 1

(b)

(c)

T2 T3

Re-structure:

Input: Z, a , b, c, and T1, T2, T3, T4

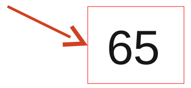
1. **Replace subtree at Z with subtree at b**
2. **Set a as left subtree of b and set T0 and T1 as left & right subtrees of a**
3. **Set c as right subtree of b and set T2 and T3 as left & right subtrees of c**

[illegible]

[illegible]

AVL Tree – Insertion – Ex. 2

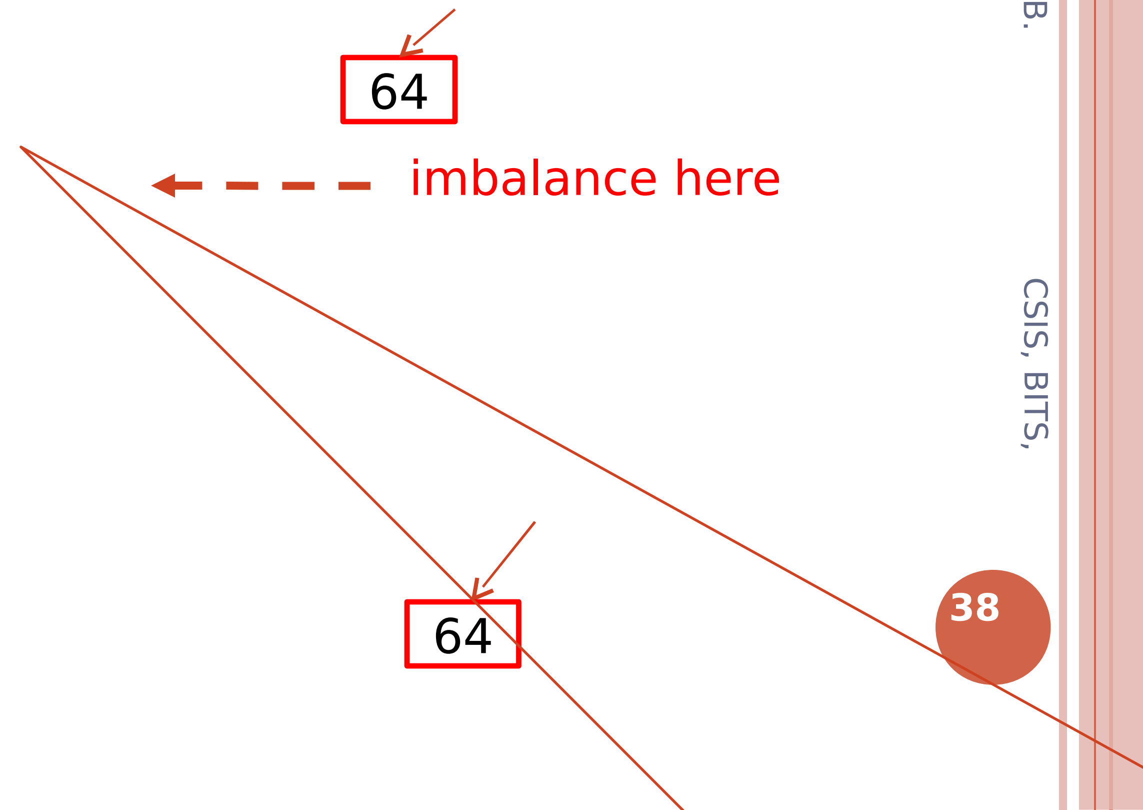
No imbalance



No rotation needed!

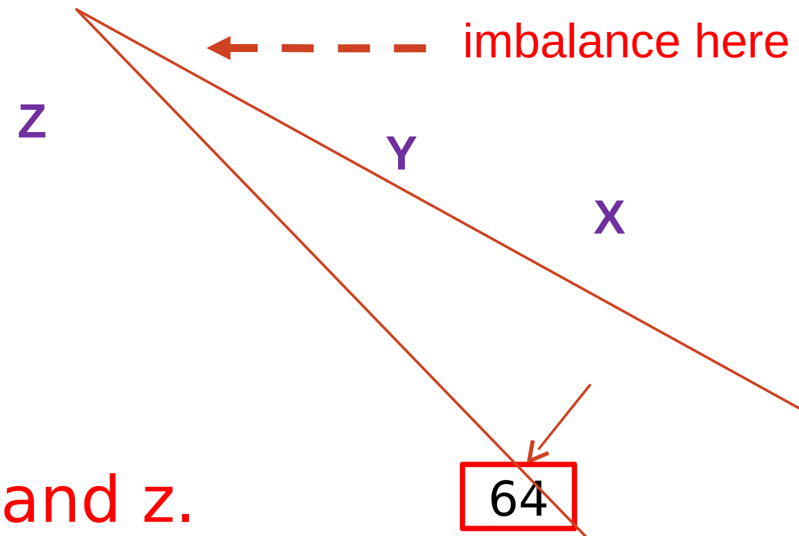
[illegible]

AVL Tree – Insertion - Cases



[illegible]

AVL Tree – Insertion - Cases



Rotate with x, y, and z.
- y will be rotated over z.

Generalized rotation:

Let Z be the first node unbalanced along the path from the inserted node to the root.

Let Y be the child of Z and X be the child of Y in the path from the inserted node to the root.

Then call rotate with X,Y, and Z

AVL Tree – ROTATION

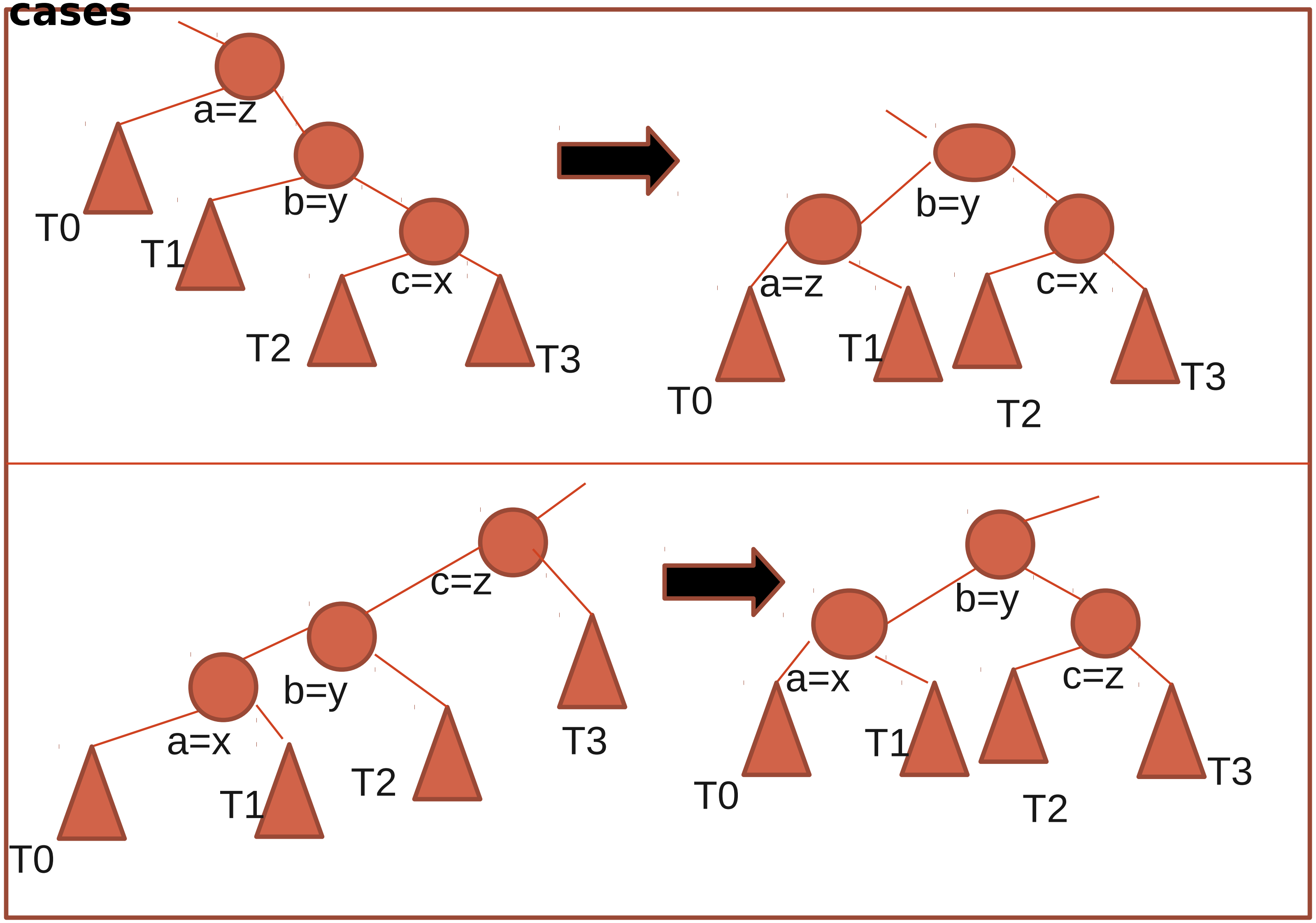
```
rotate (X, Y, Z)
{
    let a, b, c be left-to-right listing of nodes X, Y, and Z
    let T0, T1, T2, T3 be left-to-right listing of other subtrees of x,y, and z  (i.e. subtrees of X, Y, and Z not rooted at x or y)
Replace  Z with b;
Set a to be left child of b;
Set T0 and T1 to be left & right subtrees of a;
Set c to be right child of b;
Set T2 and T3 be left & right subtrees of c;
}
```

AVL Tree - Rotation

- The restructuring procedure is referred to as a rotation:
  - “geometric” visualization
- If b==Y then restructuring is referred to as a single rotation
  - i.e. rotating Y over Z
- If b==x then restructuring is referred to as a double rotation
- if b==z?
  - Argue that this case cannot happen
- Exercise: Draw templates for each possible case. How many of them are there?

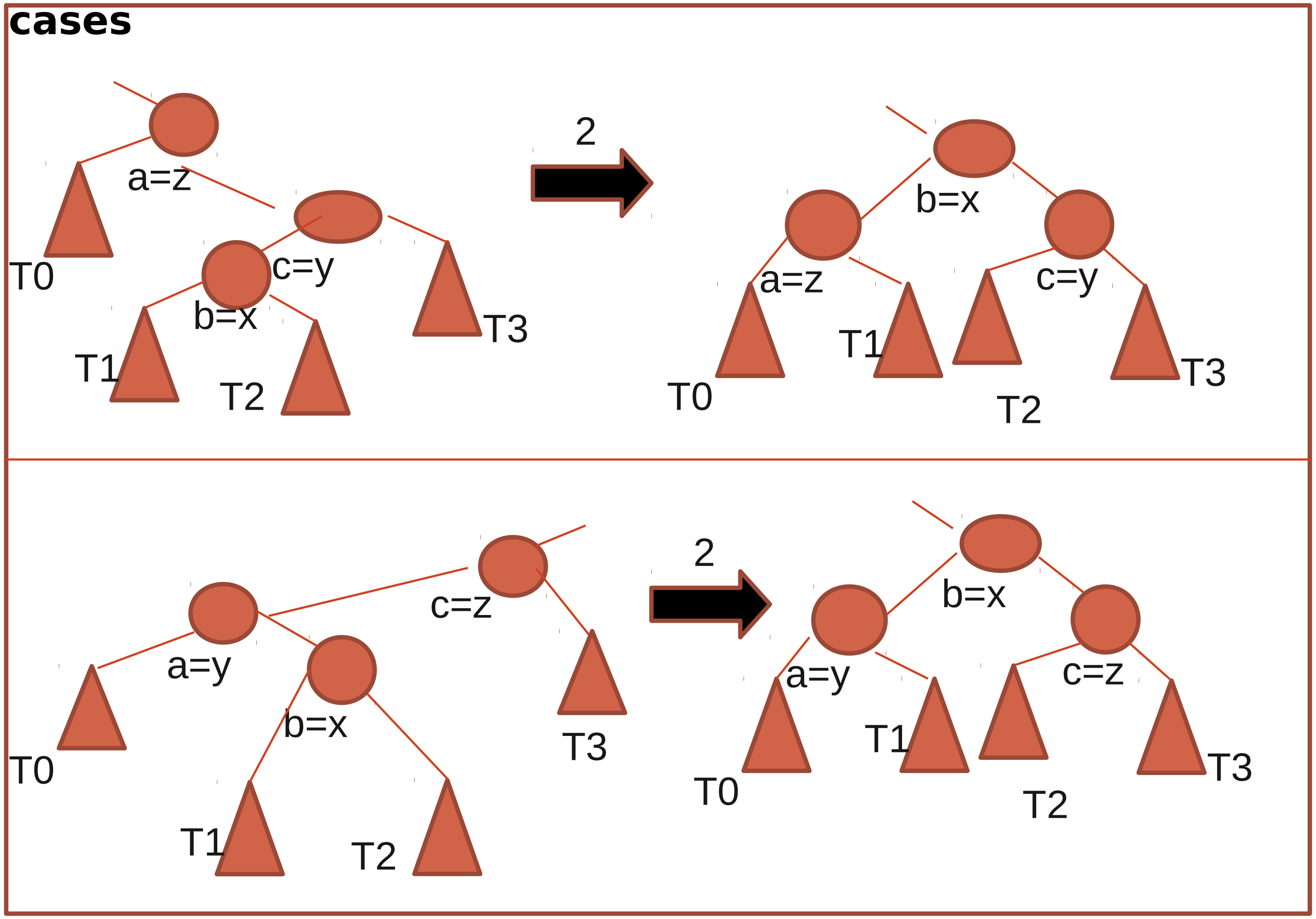
AVL Rotation Cases – Single Rotation

b=y : 2 cases



AVL Rotation Cases – Double Rotation

b=x : 2 cases



AVL Tree - Deletion

- After deletion of node W, – if W is internal, pull one of its children up (as in binary search tree).
  - This may result in imbalance (at some ancestor of W)
- Restructuring:
  - Z : first unbalanced node on the path from the deleted node to the root.
  - Y : child of Z with larger height (it won't be an ancestor of W)
  - X : child of Y with larger height (break ties arbitrarily).
  - Then call **rotate(X,Y, Z)**
- Claims:
  - This balances the node Z (locally) – Why?
  - This does not the balance the tree (globally) – Why?

AVL Tree - Deletion

- After deletion of node W:

1. if W is internal, pull one of its children up (as in binary search tree).
2. Let Z be the first unbalanced ancestral node on the way up. Balance Z by rotation.
3. Repeat step 2 until the root is balanced.

AVL Tree – Time Complexity

- Time Complexity of
  - Find:
  - O(h) and h is log N
  - Insert:
  - O(h) for finding the right position and O(1) for rotation
  - Total time is O(log N)
  - Delete:
  - O(h) for finding the right node (to be deleted) and O(h) rotations, each rotation taking time O(1).
  - Total time is O(log N)

AVL Trees – Implementation Issues

- How do we check for an unbalanced node?
  - Every node maintains a (relative) weight:
  - 0 ==> balanced
  - 1 ==> right sub tree is taller
  - -1 ==> left sub tree is taller
  - On insertion:
  - Weights are to be updated
  - If insertion happens on the right sub tree of node with weight 1 then it *may become unbalanced*
  - Similarly for a left sub tree of node with weight -1

Dictionary - Comparison

Balanced BST	Hashtable
Time Complexity: - ϴ(logN) - worst case and average case	Time Complexity: - ϴ(1) average case and ϴ(N) worst case
Space Complexity - ϴ(N) links, - ϴ(N) space for counts (height balance info.)	Space Complexity - ϴ(N) words – separate chaining (Table and links) - ϴ(N) bits – empty/non-empty

AVL Tree –Complexity

- Despite the improved time complexity, Hashtables are preferred to AVL trees in practice:
  - Most often hashtables behave well – O(1) operations with high probability
  - Implementation is complex for AVL trees
  - Rotations in AVL tree destroy locality of memory references.*
    - Why? [Consider the pointer / subtree changes.]
    - Affects caching / paging behavior resulting in bad performance.
  - Update of height balance information results in dirty caches / pages *
    - Virtual Memory performance suffers

*** See Notes on Memory Hierarchy (at the end of this slide set)**

AVL Tree –Complexity [2]

- AVL Trees are preferred only if
- bound O(log N) is strictly needed OR
- Ordered operations are needed.
  - E.g. find the minimum element
  - find all elements with key < K in order

CS C341 / IS C361
Data Structures & Algorithms

Partially Ordered Data

Totally Ordered Data vs. Partially Ordered Data
- ADT Priority Queue

Data Structure for Priority Queue
- A Heap
  - Representation (Binary Trees and Arrays)
  - Implementation of Operations

Other Applications of Heaps
- Heap Sort: Algorithm, Complexity, Comparisons
- Multi-way Merge

Partial Order

- A (binary) relation R, subset of S x S, is said to be a *partial order* if
  - (it is reflexive: x R x for all x in S)
  - it is transitive: x R y and y R z implies x R z for x,y,z in S
  - it is anti-symmetric: x R y and y R x implies x = y
- Example
  - “is an ancestor of” on persons
  - “under-writes” on companies

Total Order

- A (binary) relation R, subset of S x S, is said to be a total order if
  - x R y OR y R x for any x and y in S.
- Examples:
  - <= on real numbers
  - “is at least as old as” on persons

Total Order vs. Partial Order

- Implications:
  - 5th element?
  - Minimum element?
  - Linear Order – Refer Discrete Structures
  - Least common ancestor
  - Join and Meet – Refer Discrete Structures
    - E.g. Tournament Trees:
      - For a Tennis tournament (or any knock-out pairing based tournament)
      - Given pairing information predict who may meet whom at a later stage

Total Order vs. Partial Order

- Application:
  - Consider a multi-processing operating system on a single processor
  - More than one process could be ready (for CPU to be available)
  - Processes are to be scheduled
    - No self knowledge or co-ordination among processes
  - Need a queue for "ready" processes
  - Need a strategy for adding/deleting processes
  - FCFS - Total Order ==> (FIFO) Queue
  - Priority – Partial Order ==> Priority Queue
    - Deadline – special case of priority

ADT Priority Queue

- A ***priority queue*** is a collection of elements partially ordered by "priority"
- ADT **Priority Queue** - Interface:
  - Element findHighestPriorityElement(PQueue)
  - PQueue deleteHighestPriorityElement(PQueue)
  - PQueue addElement(PQueue)

ADT Priority Queue

- Terminology
  - Typically, *find* and *delete* are referred to as *findMin* and *deleteMin*
  - or alternatively as findMax and deleteMax
  - The notion of “Highest priority” is independent of the representation of “priority”
  - E.g. if priority is denoted by natural numbers, one may choose
    - 0, 1, 2, ...   to be decreasing order of priority.

ADT Priority Queue - Representation

- Can we use "totally ordered" data structures for implementation?
  - E.g. Binary Search Trees or Sorted Arrays
  - Implementation and Time Complexity
- Can we use "un-ordered" data structures for implementation?
  - E.g. Hashtable
  - Implementation and Time Complexity ?
- Can we find a simpler representation?
  - Can we implement operations more efficiently?
- Clue:
  - Total ordering is not needed
  - Can we balance the tree in some other fashion?

ADT Priority Queue – Representation

- (Min)-Heap Order Property (for a binary tree T):
  - For each node u and its children v1 and v2,
  - *u.key <= v1.key* && u.key <= v2.key
- A *complete binary tree* that satisfies the heap order property is said to be a *heap*.
  - A binary tree of N nodes is said to be complete
    - if all (node) positions numbered 1 to N are occupied where the numbering proceeds top-down, left-to-right.
  - [alternatively: if the height is h, all levels except h have maximum number of nodes, and at level h, all nodes are (continguosly) to the left.]

Heaps

- Properties of a heap:
  - It is height balanced.
  - Implications: ??
  - It can be stored in an array of size <= $2^h - 1$
  - How do you implement
  - getLeft and getRight?
  - Given node index j: (index starts at 0)
    - 2 * j + 1 and 2 * j + 2 denote the left and right children (i.e. their indices)

Heaps

- Implementation of Operations:
  - Element find(MinHeap)
  - Trivial to implement
  - MinHeap delete(MinHeap)
  - Copy the last element to the root
  - Reduce size by 1
  - Re-order to preserve Heap-Property
  - MinHeap insert(MinHeap, Element)
  - Exercise!

Heaps - Heapify

- Precondition:
  - Sub-trees rooted at 2*t+1 and 2*t+2 are min-heaps
- Postcondition:
  - Tree rooted at t is a min-heap

```
minHeapify(Element H[], int size, int t)
 {  L = 2*t+1;   R = 2*t+2;
    if (L < size) {
          mIx = minIndex(H, L, t);
          if (R < size)  {  mIx = minIndex(H,R , mIx); }
    } else {
         mIx = t;
    }
     if (mIx <>t) { swap(mIx, t);  minHeapify(H, size, mIx);  }
 }
```

Heap – Building a Heap

- Given any array, a heap can be constructed by repeated invocations of *minHeapify*:
  - Postcondition: H is a heap

```
buildMinHeap(Element H[], int size)
{
    for (j = size/2; j>=  0; j--)   minHeapify(H, size, j);
}
```

- Proof: (by Induction)
  - Base: Singletons are heaps (i.e. H[size/2+1] … H[size])
  - Step: *minHeapify* builds a heap at level k from two heaps at level k+1

Heap

- Time Complexity of
  - minHeapify:
  - h steps, where h is the height of the heap
  - h is ┌ log N ┐ for a complete binary tree.
  - buildMinHeap:
  - In an N element heap there are at most ┌ N/2h+1 ┐ nodes of height h.
  - Total cost of buildHeap is
  - h=0 └ logN ┘ (┌ N/2h+1 ┐ * h)
  - <= N * (h=0 └ logN ┘ (h/2h))
  - <= N * 2

HeapSort

- Post-condition: A is sorted in decreasing order
- HeapSort(Element A[], int size)

```
buildMinHeap(A);
for (j=size-1; j>0; j--) {
        swap(A[0],A[j]);
        size=size-1;
        minHeapify(A,size,0);
}
```

- Time Complexity of Heapsort (for an array of size N)
  - O(N) for buildHeap + N * O(log N) for N heapify calls
  - i.e. O(N logN) in the worst case

HeapSort –Evaluation– Order Complexity

QuickSort	MergeSort	HeapSort
O(logN) space – worst case / average case	O(N) space – worst case/ average case	O(1) space – worst case / average case
O(N*N) time – worst case	O(NlogN) time – worst case	O(NlogN) time – worst case
O(NlogN) time – average case (w. high probability)	O(NlogN) time – average case	O(NlogN) time – average case

HeapSort – Evaluation – Implementation Measurements

Knuth's Measurements (MIX)

QuickSort	HeapSort
6*N*log(N) comparisons on the average	12*N*log(N) comparisons on the average
close to N*N comparisons in the worst case	18*log(N) + 38*N comparisons in the worst case

Heap – another application – Multi-way Merge

- Sorting large data sets – often stored on secondary media (tape/disk)
  - Sort subsets and merge
  - # subsets s = ceil(N/M)
    - Input set size is N , RAM size is M
  - # merge operations: s-1
  - Alternative for (traditional 2-way) Merging
  - Multi-way merging: i.e. merge k files at a time

Heap – another application – Multi-way Merge [2]

- (Multi-way Merging) Implementation:

```
for j = 0 to k-1  H[j] = read(file[j]);
buildHeap(H);
repeat {
        nextMin= find(H); H = delete(H);
        add nextMin.value in the merged list;
        if  (!empty(nextMin.file))  {
            next = read(nextMin.file);
            H = insert( H, next);   }
} until (H is empty)
```

Heap – Operations - Insertion

- Implementation:

```
• insert(Heap H, Element e)
• {
•      cur = last+1; H[cur] = e;
•       par = (cur-1)/2;
•       while ((cur>0) && (H[par] > H[cur]) {
•            swap(H, par, cur);
•            cur = par;
•            par = (par-1)/2;
• }
```

CS C341 / IS C361 Data Structures & Algorithms

Binary Trees
- Traversal(s) and Applications

Binary Tree – Review

- Definition: A Binary Tree is either
  - an empty Binary Tree OR
  - has a root value and two (sub) Binary Trees.
- Type Definition
  - BinaryTree = EmptyBinaryTree U

    (Element * BinaryTree * BinaryTree)
- Representation (in C)
  - typedef struct _binTree *BinTree;
  - struct _binTree {

    Element val; BinTree left, BinTree right;

};

Binary Tree – Review [2]

- BinaryTree - Operations
  - BinTree createBinTree()
  - boolean isEmptyBinTree(BinTree)

- Properties:
  - isEmptyBinTree(createBinTree()) == TRUE

Binary Tree – Review [2]

- BinaryTree - Operations
  - BinTree left(BinTree)
  - BinTree right(BinTree)
  - Element rootVal(BinTree)
  - BinTree makeBinTree(Element, BinTree, BinaryTree)
- Properties:
  - makeBinTree(rootVal(bt), left(bt), right(bt)) == bt

Binary Tree - Traversals

- Typical Requirements for a traversal:
  - Enumerating the elements in a collection (represented as a binary tree)
  - Applying some function / procedure on each element in a collection (represented as a binary tree)
- Order of traversal
  - In-Order Traversal:
    - Traverse left, visit Root, Traverse right
  - Application:
    - Enumeration in sorted order in a BST
      - Left – Right vs. Right – Left ??

* • *+5

Binary Tree - Traversals

- Consider an expression of the form:
  - (* (* 3 4) (+5 7))
  - Referred to as a “prefix” expression.
- Convert this into an internal representation:

Q: What is the difference between these two forms of representation?

Binary Tree - Traversals

- How do you construct such a representation?
  - Construct the root Node
  - (* (* 3 4) (+5 7))

* • * 3

Binary Tree - Traversals

- How do you construct such a representation?
  - Construct the root Node
  - Construct the left sub-tree (i.e. left sub-expression)

  - (* (* 3 4) (+5 7))

* • * + 5

Binary Tree - Traversals

- How do you construct such a representation?
  - Construct the root Node
  - Construct the left sub-tree
  - Construct the right sub-tree (i.e. right sub-expression)
  - (* (* 3 4) <u>(+5 7))</u>

Binary Tree - Traversals

- Pre-Order Traversal:
  - visit Root, Traverse left, Traverse right
- Question:
  - Does left-to-right order matter?
  - e.g. Construction of a binary search tree
- Special case:
  - *find* operation in a BST

* • + 5

Binary Tree - Traversals

- How do you evaluate an expression - given a tree representation?
  - Evaluate the left sub-tree
  - Evaluate the right sub-tree
  - Evaluate the root

- Post-Order Traversal:
  - Traverse left, Traverse right, visit Root

Binary Tree – Application - Encoding

- Encoding Problem:
  - Consider a scenario where strings of symbols are to be encoded:
  - e.g. Machine instructions (*opcodes, addresses*)
  - e.g. Binary representation of HTML/XML documents

```
<BOOKS>
<BOOK YEAR="1999">
<AUTHOR>Abiteboul</AUTHOR>
<AUTHOR>Buneman</AUTHOR>
<TITLE>Data on the Web</TITLE>
<PRICE>40.00</PRICE>
<SHIPPING>10.00</SHIPPING>
</BOOK>
<BOOK YEAR="2002">
...
</BOOK>
</BOOKS>
```

Binary Tree – Application - Encoding

- Encoding Technique
  - If you have N different “symbols” to be encoded,
  - then ┌ logN ┐ bits are required to encode each occurrence of each item
  - *fixed length binary coding*
- Given a string of length M where each item may be any of the N “symbols”
  - Size of the representation is M* ┌ logN ┐
  - Decoding each item (from the encoded form) requires inspecting all the ┌ logN ┐ bits.
- Is it possible to reduce the number of bits required or the work required to decode?

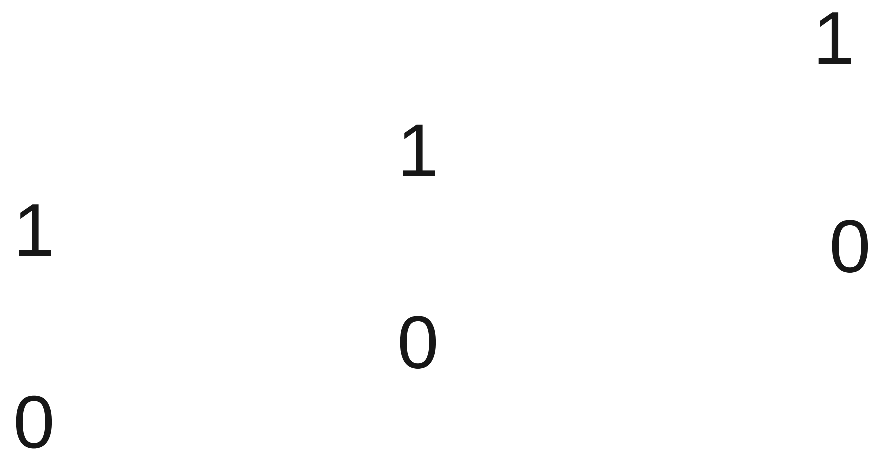
Binary Tree – Application - Encoding

- Encoding Technique
  - Consider the frequency of occurrence of those symbols:
  - e.g. AUTHOR may occur more often than other symbols in the particular XML database
  - e.g. ADD is the most common instruction in most programs.
  - Encode the most common symbol as the shortest code (1 bit):
  - Say, ADD is encoded as 0
  - Then1 would represent “Any symbol other than ADD”
  - Encode the next most frequent symbol as 10
  - …
  - Variable length coding
  - Specifically known as Prefix codes
  - Size of representation = ∑ freq(c) * encLen(c)

A L C
D O M
D A P
D

Binary Tree – Application - Encoding

- How does decoding work?
  - Say, ADD is encoded as 0, LOAD is encoded as 10, CMP is encoded as 110, and JMP is encoded as 111



- Each code is a path from the root to a leaf in the tree

Binary Tree – Application - Encoding

- Huffman Coding Technique:
    - Produces optimal prefix code given frequencies of items (to be coded)
- Preconditions : C is an array of symbols;
-                          for each c in C, c.freq is the frequency of the symbol
- Output: Decoding tree for C

```
HuffmanCode(C) {
H = buildHeap(C);   // H is C after heapification!
for j = 1 to |C|-1 {
        x = find(H); H = delete(H);
        y = find(H); H = delete(H);
        H = insert(makeBinTree(x. freq + y.freq, x, y), H);
}
return find(H)
}
```

Binary Tree – Application - Encoding

- Huffman’s encoding algorithm produces optimal prefix code:
  - Proof omitted.
- Huffman’s encoding algorithm uses a “greedy” technique:
  - It makes “a local (i.e. greedy) choice” that results in “ a globally optimal” solution.
  - Choice of two lowest frequency items to have the longest code(s).
- Greedy Technique is a design technique to produce efficient algorithms.
  - [Will see more of it later!]