COEN 12 Midterm #2 Study Guide

Linked Lists/Binary Trees

1. Linked Lists: a linked list consists of a series of nodes which each contain data
   1. Types of linked lists:
      1. Singly-linked list
         1. Each node has data and a link to the next node in the list
      2. Doubly-linked list
         1. Each node also has a link to the previous node as well as the next node
      3. Circular-linked list
         1. Circular singly-linked list
            1. Tail has a link to the head
         2. Circular doubly-linked list
            1. Head has a link to the tail as well
         3. In addition to maintaining a head pointer, we also an maintain a tail pointer
      4. The key difference between a linked list and an array is the layout in memory:
         1. Array slots are in consecutive memory locations
         2. Nodes in a linked list can be any where in memory but we need the extra storage of a link to get to the next node
      5. Linked list in C
         1. To keep the link and data together we use a struct
         2. For just a node:

struct node

{

int data;

struct node \*next;

};

* + - 1. For a list of nodes

{

int count;

struct node \*head;

};

* + - 1. Note that binary search on a linked list, while technically possible; is not practical. Why? Arrays support indexing in O(1) time; linked lists do not.
      2. Sequential search for linked list algorithm

struct node \*pcur;

pcur = pList->head;

while(pcur!=NULL)

{

if (pcur->data == data)

return true;

pcur = pcur->next;

}

* 1. Linked list insertion
     1. Four cases to consider for linked lists
     2. Things to note:
        1. Rather have many things pointed to something rather than none
        2. Do node pointers first then the node
        3. Insert after the previous node
        4. Empy list

{

pNew->next = NULL [or pList->head]

pList->head = pNew;

pList->count++;

}

* + - 1. At the front

{

pNew->next = pList->Head;

pList ->head = pNew;

pList ->count++;

}

* + - 1. At the rear

{

pNew->next = NULL [pPrev->next or pTail->next];

pTail->next [or pPrev->next] = pNew;

pList->count++;

}

* + - 1. In the middle

{

pNew->next = pPrev ->next;

pPrev->next=pNew

pList->count++;

}

* + 1. Our algorithm for insertion can be reduced to 2 main cases
       1. Inserting as the first node
       2. Inserting somewhere else
    2. Able to reduce algorithm even further to 1 case with dummy (sentinel) node
    3. Dummy node: is a special node at the start of the list. It doesn’t hold valid data but merely always marks the start of the list
       1. In a list with a dummy node, the head pointer never points to NULL. We eliminate the special case of insertion as the fist node in the list. To inset a new data node as the first data node in the list, we insert AFTER the dummy node

{

pNew->next = pPrev ->next;

pPrev->next = pNew;

}

* + 1. Insertion Recap
       1. Insertion in a linked is O(1) if we are given the place to insert. If we have to find the proper location then it is O(n)
  1. Linked List Deletion
     1. Three cases for deletion
     2. Previous pointer and pointer to the node being deleted needed for deletion
        1. Deleting the first node

{

pDel = pList->head;

pList->head = pList->head->next;

free(pDel);

pList->count--;

}

* + - 1. Deleting the last node

{

pDel = pPrev->next;

pPrev->next = NULL;

free(pDel);

pList->count--;

}

* + - 1. Deleting from the middle

{

pDel = pPrev->next;

pPrev->next = pDel->next;

free(pDel);

pList->count--;

}

* + - 1. Tail and middle deletion the same
      2. Pointer directly to the node being deleted is useless because the pointer to the rest of the nodes will be lost when deleted
    1. If we employed a dummy node for the first case (handled by the if part) would go away
    2. Regardless, deletion is O(1) once we know which node to delete
    3. Check will be needed on exam to make sure code runs correctly in all scenarios

1. Botany 101 (Trees)
   1. Trees: we can think of a linked list as a collection of nodes in which each node has at most one predecessor but can have an arbitrary number of successors
      1. We can think of a tree as a collection of nodes in which each node has at most one predecessor but can have an arbitrary number of successors
      2. Grow down from top bottom
   2. Tree Terminology
      1. Root: has no predecessors; nodes that have no parent
      2. Leaves: tails of the list of nodes; nodes that have no children
      3. Parent: predecessor
      4. Children: following nodes
      5. Recursive definition of a tree: empty or a node with children all of which are subtrees
      6. Forest: Set of trees
      7. Linked lists are trees but not all trees are linked lists
   3. Tree Traversals
      1. Preorder: root of each subtree is printed or visited before its children
         1. Top-down
         2. Left to right
         3. Farthest right will be printed last

{

void preorder(struct node\*np)

{

if (np!=NULL)

{

printf(“%d\n”, np->data);

preorder(np->left);

preorder(np->right);

}

}

}

* + 1. Postorder: root is visited after all of its children
       1. Bottom-up
       2. Left to right
       3. Root will be printed last

{

postorder (struct node\*np)

{

if (np!=NULL)

{

postorder(np->left);

postorder(np->right);

printf(“%d\n”, np->data);

}

}

}

1. Binary Tree: a binary tree is a tree in which each node has at most two children
   * 1. Two children are referred to as a left child and right child
        1. Implementing a Binary tree

{

int data;

struct node \*left, \*right;

}

* 1. Height of the Tree: is a tree defined as the length of the longest path from the root to a leaf
     1. Number of nodes reached including first
     2. Number of nodes in tree exclude the initial node (root)
     3. Given a binary tree of height h, what are the minimum and maximum number of nodes in the tree
        1. O(h)<=n<=O(2^h)
        2. h: minimum number of nodes stretched out
        3. 2^h – 1: maximum number of nodes compress for max number of nodes
     4. Binary tree of n nodes, what are its minimum and maximum height
        1. O(logn)<=h<=n
        2. Logn: Compressed structure with multiple nodes for a reduced height
        3. n: stretched out to be max height
  2. Binary Search Tree: is a binary tree that obeys the search tree property: for each node in the tree all nodes in the left subtree are smaller than the root and all nodes in the right subtree are greater than the root
     + 1. Typical run time of O(h),
          1. Go longest path, the height of the tree
     1. Search a binary tree we use binary search

{

bool bsearch (struct node \*np, int x)

{

while (np!=NULL)

{

if (x<np->data)

np = np->left;

else if (x>np->data)

np = np->right;

else

return true;

}

return false;

}

}

* + 1. Recursive search

{

bool bsearch(struct node\*np, int x)

{

if (np == NULL)

return false;

if (np->data < x)

return bsearch (np->left, x);

if (np->data > x)

return bsearch (np->right, x);

return true;

}

}

* + 1. Recursive minimum

int minimum (struct node \*np)

{

assert (np!=NULL);

if (np->left==NULL)

return np->data;

else

return minimum (np->left);

* + - 1. While statement could replace if for a nonrecursive function
    1. Transversing BST
       1. Preorder (top-bottom): self, left, right
       2. Postorder (bottom-top): left, right, self
       3. Inorder: left, self, right
          1. Inorder printing:

Void inorder (struct node \*np)

{

if (np!=NULL)

inorder (np->left);

printf (“%d\n”, np->data);

inorder (np->right);

}

* 1. Inserting Values into a BST
     1. When inserting a value into a BST, we always inset the new value as a leaf
     2. We have to place the new value in the correct place such that if we were to later search for it, we would find it
        1. First number from set of numbers is the root
     3. We won’t prove this fact, but on average a BST hast height proportional to logn
  2. Deleting values from a BST
     1. When deleting values from a BST, we will not always be deleting a leaf. We might be deleting a node with children
     2. Lets consider 3 cases:
        1. Case 0: no children (leaf)
           1. To delete a node with no children, we can simply set its parent’s child’s pointer to NULL
           2. Then we can delete (free) the node
        2. Case 1: one child
           1. To delete a node with only one child, we simply have our parent’s child pointer point to the child rather than the node being deleted
           2. Then we can delete (free) the node
        3. Case 2: two children
           1. To delete a node with two children, WE DON’T
           2. INSTEAD, we transform the tree so that we will have a case 0 or case 1
           3. We first find either the min (smallest value) of the right subtree or the max (biggest value) of the left subtree (it doesn’t matter)
           4. We replace the node being deleted with the value of the node found. This will not break the search
           5. We do however now have two copies of the value of the node found. So we now delete the original value from its subtree. This deletion will either be case 0 or case 1
     3. Postorder most effecint way to delete leafs in a tree
     4. NEVER delete a node with two children
     5. Watch out for proper case scenarios
  3. Recap of BST
     1. We have seen that min, max, search, insert, and delete all run in O(h)
     2. The problem with BST’s as we have defined them is that they can become unbalanced
     3. We say that a tree is balanced if a path from the root to a leaf is not “much farther” than any other path from the root to a leaf

1. AVL Tree: is a BST tree if for each node in the tree, the heights of its left and right subtrees differ by no more than one
   * 1. |Hl – Hr| =< 1
     2. we can label each node in an AVL tree as either:
        1. LH: left high
        2. EH: even high
        3. RH: right high
     3. Since an AVL tree is a BST, we don’t have to change min, max, or search because these operations don’t change the trees
     4. However, insert and delete will need to change
   1. Inserting a node into an AVL tree, we complete two steps:
      1. We insert the new node as a leaf just like we did before
      2. Doing so will not mess up the search tree property. However, the tree could now become unbalanced
      3. We go back up the path we came down, and if we find that is now unbalanced we rebalance it
      4. Rebalancing is done through rotations:
   2. 4 cases of insertion
      1. Edit tree as little as possible when rotating
      2. Case 1: Left of left: a left high subtree of a previously left high subtree
         1. Steps required:
            1. Rotate the unbalanced node to the right
            2. The left child of the unbalanced node becomes the new root of the subtree
            3. Any right child of the new root (shifted child root) become the left child of the old root (moved root)
      3. Case 2: right of right (symmetrical to case 1)
      4. Case 3: Right of left
         1. Steps required:
            1. Rotate the left child of the unbalanced node to the left
            2. This will result in case 1
            3. Rotate the unbalanced node to the right
      5. Case 4: left of right (symmetrical to case 3)
      6. (Side/direction of unbalance) of (unbalanced root/subtree)
      7. Insert, min, max, search (and delete) all still run in O(h) steps, but since the tree is balanced, h is O(logn)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Set | Unsorted Array  (sequential search) | Sorted Array  (Binary search) | Hash Table | Unsorted Linked List | Sorted Linked List | BST | AVL |
| Find | O (n) | O (logn) | O (1) /  O (n) | O (n) | O (n) | O (logn) / O(h)= O(n) | O(logn) |
| Add | O (n)  Bag:  O (1) | O (n) | O (1) /  O (n) | O (n)  Bag:  O (1) | O (n) | O (logn) / O(h) = O(n) | O (logn) |
| Remove | O (n) | O (n) | O (1) /  O (n) | O (n) | O (n) | O (logn) / O(h) = O(n) | O (logn) |
| Min/Max | O (n) | O (1) | O (m) | O (n) | O (1)  (Depends on tail pointer) | O (logn) / O(h) = O(n) | O (logn) |
| Remove all | O (n) | O (n) | O (m) | O (n) | O (n) | O(logn) /  O(h) = O(n) | O (logn) |
| Count number of occurrences | O (n) | O (n) | O (m) | O (n) | O (n) |

Best to worst case