Midterm 2 review CSCI 2270

1. How does a binary search tree’s shape depend on the order of the numbers inserted into it?
2. What parts are similar in the two processes of searching a binary tree and searching a sorted array by binary search? What parts are different? What depends on luck?

An array allows random access to each element in it. so you get insert, delete and look for a specific element in O(1), and max/min, delete in O(n). [you can also make max/min O(1) and delete O(n) instead]. If you are keeping your array sorted, it will cause insert/delete to be O(n), but you will gain O(logn) find, and O(1) min/max.

A BST is sorted by definition, and for a regular [unbalanced] BST, you get O(n) worst case behavior. For balanced BST, you get O(logn) insert/delete/find. You can get O(1) min/max any how for both.

Arrays are also usually faster to iterate [assuming order of iteration is not important] since you gain better cache performance. Also, unlike BST - which has unbounded size by nature, an array requires reallocation and copying the data when your array is full.

Improving a BST can be done by making it balanced - like AVL or red-black-trees.

Which is better? it depends on the application. Usually when you are planning to insert data and keep it sorted, BST will be prefered. If random access or iteration is the main purpose: you usually use an array.

1. Given an arbitrary binary tree, print it out in preorder, inorder, and postorder.

|  |  |  |  |
| --- | --- | --- | --- |
| http://upload.wikimedia.org/wikipedia/commons/thumb/d/d4/Sorted_binary_tree_preorder.svg/220px-Sorted_binary_tree_preorder.svg.png | http://upload.wikimedia.org/wikipedia/commons/thumb/7/77/Sorted_binary_tree_inorder.svg/220px-Sorted_binary_tree_inorder.svg.png | | http://upload.wikimedia.org/wikipedia/commons/thumb/9/9d/Sorted_binary_tree_postorder.svg/220px-Sorted_binary_tree_postorder.svg.png |
| Preorder | Inorder | Postorder | |
| void Tree::preOrder(Node\* n) {  if ( n ) {  cout << n->Key() << " ";  preOrder(n->Left());  preOrder(n->Right());  }  } | void Tree::inOrder(Node\* n) {  if ( n ) {  inOrder(n->Left());  cout << n->Key() << " ";  inOrder(n->Right());  }  } | void Tree::postOrder(Node\* n) {  if ( n ) {  postOrder(n->Left());  postOrder(n->Right());  cout << n->Key() << " ";  }  } | |

1. Given a bunch of numbers, in some order, insert them into a binary search tree.

|  |  |
| --- | --- |
| 1. check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise, 2. if a new value is less, than the node's value: 3. if a current node has no left child, place for insertion has been found; 4. otherwise, handle the left child with the same algorithm. 5. if a new value is greater, than the node's value: 6. if a current node has no right child, place for insertion has been found; 7. otherwise, handle the right child with the same algorithm. | void insert(Node\*& root, int data) {  if (!root)  root = new Node(data);  else if (data < root->data)  insert(root->left, data);  else if (data > root->data)  insert(root->right, data);  } |

5a. Given the binary search code and a particular array of sorted numbers, tell me the first array slot the search code will check to find 3 in the array 1 3 5 6 8 9 11 14. What’s the last array slot a search for the 3 will check?

First Search: (0+7) / 2 = 3.5 = array[3]  
 Second Search: (0+2) / 2 = 1 = array[1]

5b. Repeat the question but look for a number that’s not in the array, like 10. What will be the last slot checked?

First Search: (0+7) / 2 = 3.5 = array[3]  
 Second Search: (4+7) / 2 = 5.5 = array[5]  
 Third Search: (6+7) / 2 = 6.5 = array[6]

1. To get the 6 big\_number comparison functions ==, !=, <, >, <=, and >=, how many must you write, and why, and what can you do for the other ones instead of writing them all from scratch?
2. What time penalty comes from using the add\_node function when copying a list?

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Linked List | Array | Dynamic Array | Balanced Tree | Random Access List |
| Indexing | O(n) | O(1) | O(1) | O(logn) | O(logn) |
| Insert/Delete at Beginning | O(1) | Na | O(n) | O(logn) | O(1) |
| Insert/Delete at end | Θ(n) when last element is unknown;  Θ(1) when last element is known | na | O(1) | O(logn) | O(logn) |
| Insert/Delete at Middle | Search time + O(1) | Na | O(n) | O(logn) | O(logn) |
| Wasted Space | O(n) | 0 | O(n) | O(n) | O(n) |

1. When is a binary search tree most efficient? Least efficient? Why?

|  |  |
| --- | --- |
| The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient. The other advantages are:   1. Binary Search Tree is fast in insertion and deletion etc. when balanced. 2. Very efficient and its code is easier than other data structures. 3. keys in the nodes in a way that searching, insertion and deletion can be done efficiently. 4. Implementation is very simple in Binary Search Trees. 5. Nodes in tree are dynamic in nature. | Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays. Some of their disadvantages are as follows:   1. The shape of the binary search tree totally depends on the order of insertions, and it can be degenerated. 2. When inserting or searching for an element in binary search tree, the key of each visited node has to be compared with the key of the element to be inserted or found, i.e., it takes a long time to search an element in a binary search tree. 3. The keys in the binary search tree may be long and the run time may increase. 4. After a long intermixed sequence of random insertion and deletion, the expected height of the tree approaches square root of the number of keys which grows much faster than \log n. |

1. Given the code in bintree.cpp, can you make a function that multiplies every number in a binary tree by 7?
2. Given the code in bintree.cpp, can you make a function that reverses (mirror images a binary search tree)?

struct node \*mirror(struct node \*here) {

if (here == NULL)

return NULL;

else {

struct node \*newNode = malloc (sizeof(struct node));

newNode->value = here->value;

newNode->left = mirror(here->right);

newNode->right = mirror(here->left);

return newNode;

}

}

1. If you had a mirror imaged binary search tree, what would you need to do when inserting data into it?

In the insert method:  
if (payload < root\_ptr->data) insert(root\_ptr->right)  
if (payload > root\_ptr->data) insert (root\_ptr->left)

1. Why is self assignment a problem for operator =?
2. Why is self assignment not a problem for the copy constructor?
3. What is the difference between an assignment operator and a copy constructor?

|  |
| --- |
| A copy constructor is used to initialize a previously uninitialized object from some other object's data. An assignment operator is used to replace the data of a previously initialized object with some other object's data. |
| The first is copy initialization, the second is just assignment. There's no such thing as assignment constructor.  A aa=bb;  uses the compiler-generated copy constructor.  A cc;  cc=aa;  uses the default constructor to construct cc, and then the *assignment operator*\* (operator =) on an already existing object. |
| * If a new object has to be created before the copying can occur, the copy constructor is used. * If a new object does not have to be created before the copying can occur, the assignment operator is used. |

1. Use pointer arithmetic to write a function to reverse an array.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Set two pointers: one to the end and one to the beginning. Then move them toward each other, swapping the contents as you go. When the two pointers point to the same place, you're done. | |  | | --- | | void reverseArray (int \*arr, int nElts) |  |  |  | | --- | --- | | 02 | { |  |  |  | | --- | --- | | 03 | int \*i,\*j,temp,\*end; |  |  |  | | --- | --- | | 04 | if(nElts>0) |  |  |  | | --- | --- | | 05 | end=arr+(nElts-1); |  |  |  | | --- | --- | | 06 | for (i=arr,j=end;j>i;j--,i++) |  |  |  | | --- | --- | | 07 | { |  |  |  | | --- | --- | | 08 | temp=\*i; |  |  |  | | --- | --- | | 09 | \*i=\*j; |  |  |  | | --- | --- | | 10 | \*j=temp; |  |  |  | | --- | --- | | 11 | } |  |  |  | | --- | --- | | 12 | } | |

1. Why can’t we do binary search on a linked list?
2. Why is contains for a binary search tree faster than O(n)? Can binary tree contains be this fast?

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's height (see tree terminology). On average, binary search trees with n nodes have O(log n) height. However, in the worst case, binary search trees can have O(n) height, when the unbalanced tree resembles a linked list (degenerate tree).

1. Suppose I am adding 2 big\_numbers as follows:

big\_number alice(98);   
big\_number bobo(87); alice+=bobo;

In the code for operator +=, big\_number& big\_number::operator+= (const big\_number& b) which number, alice or bobo corresponds to b? Which number corresponds to \*this?

1. Tell me how a stack can be used to tell if a program has balanced {}.

1) Declare a character stack S.

2) Now traverse the expression string exp.

a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[') then push it to stack.

b) If the current character is a closing bracket (')' or '}' or ']‘) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.

3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

1. Trace out the tree\_copy function for a particular binary tree. Which node is copied first? Last?

int minValue(struct node\* node) {

  struct node\* current = node;

  /\* loop down to find the leftmost leaf \*/

  while (current->left != NULL) {

    current = current->left;

  }

  return(current->data);

}

1. Trace out the tree\_clear function for a particular binary tree. Which node is cleared first? Last?

5 Order: 2, 4, 3, 6, 9, 7, 5

3 7

2 4 6 9

1. Be nauseatingly familiar with the copy command.
2. Where is the smallest number in a binary search tree? How would you find it?

Left most leaf node.  
int minValue(struct node\* node) {

  struct node\* current = node;

  /\* loop down to find the leftmost leaf \*/

  while (current->left != NULL) {

    current = current->left;

  }

  return(current->data);

}

1. When I compare 2 big\_numbers, which digits should I compare first and why?