Lecture 8-2 - Data Structures

Binary Search Trees

Implementing Binary Trees: class binaryTreeType Functions

Public	**Private**
isEmpty	copyTree
inorderTraversal	Destroy
preorderTraversal	Inorder, preorder, postorder
postorderTraversal	Height
treeHeight	Max
treeNodeCount	nodeCount
treeLeavesCount	leavesCount
destroyTree	

Binary Search Trees

- *Data* in each *node*
  - *Larger* than the data in its *left child*
  - *Smaller* than the data in its *right child*
- A **binary search tree**, ***t***, is either *empty* or:
  - *T* has a special node called the ***root*** *node*
  - *T* has *two sets of nodes*, L_T and R_T, called the *left subtree* and *right subtree* of *T*, respectively
  - *Key* in *root* node *larger* than every *key* in *left subtree* and *smaller* than every *key in right subtree*
  - L_T and R_T are *binary search trees*

Binary Search Trees

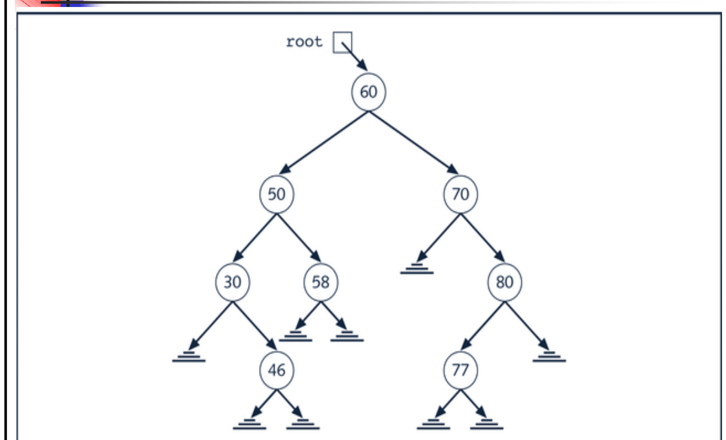


Figure 11-10 Binary search tree

Operations Performed on Binary Search Trees

- Determine whether the binary search tree is *empty*
- *Search* the binary search tree *for a particular item*
- *Insert* an item in the binary search tree
- *Delete* an item from the binary search tree

5

Operations Performed on Binary Search Trees

- Find the *height* of the binary search tree
- Find the *number of nodes* in the binary search tree
- Find the *number of leaves* in the binary search tree
- *Traverse* the binary search tree
- *Copy* the binary search tree

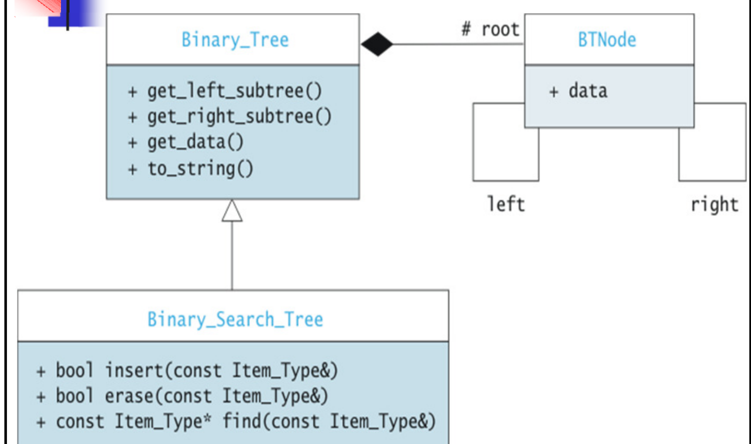
6

Searching a Binary Tree: Algorithm

```
if root is NULL
    item not found in tree: return NULL
compare target and root->data
if they are equal
    target is found, return root->data
else if target < root->data
    return search(left subtree)
else
    return search(right subtree)
```

7

Design of Binary_Search_Tree



8

Definition of Binary_Search_Tree Class

```
template<typename Item_Type>
  class Binary_Search_Tree : public Binary_Tree<Item_Type>
{
  public:
  // Constructor
  Binary_Search_Tree() : Binary_Tree<Item_Type>() {}

  // Public Member Functions
  virtual bool insert(const Item_Type& item);

  virtual bool erase(const Item_Type& item);

  const Item_Type* find(const Item_Type& target) const;
```

9

Binary_Search_Tree Definition (2)

```
private:

  // Private Member Functions
  virtual bool insert(BTNode<Item_Type>* &local_root,
          const Item_Type& item);

  virtual bool erase(BTNode<Item_Type>* &local_root,
          const Item_Type& item);

  const Item_Type* find(BTNode<Item_Type>* local_root,
                    const Item_Type& target) const;

  virtual void replace_parent(
                   BTNode<Item_Type>* &old_root,
                   BTNode<Item_Type>* &local_root);

};        // End binary search tree
```

10

The `Find` Function

```
template<typename Item_Type>
  const Item_Type* Binary_Search_Tree<Item_Type>::find(
    const Item_Type& item) const {
  return find(this->root, item);
}

template<typename Item_Type>
  const Item_Type* Binary_Search_Tree<Item_Type>::find(
    BTNode<Item_Type>* local_root,
    const Item_Type& target) const {
  if (local_root == NULL)
    return NULL;
  if (target < local_root->data)
    return find(local_root->left, target);
  else if (local_root->data < target)
    return find(local_root->right, target);
  else
    return &(local_root->data);
}
```

11

Insertion into a Binary Search Tree

Binary Search Tree Add Algorithm

1. if *root* is **NULL**
2. *replace* empty tree with *new data leaf*; return ***true***
3. if ***item equals root->data***
4. return ***false***
5. if ***item < root->data***
6. return *insert(left subtree, item)*
7. else
8. return *insert(right subtree, item)*

12

The `insert` function

```
template<typename Item_Type>
  bool Binary_Search_Tree<Item_Type>::insert(
    const Item_Type& item) {
  return insert(this->root, item);
}

template<typename Item_Type>
  bool Binary_Search_Tree<Item_Type>::insert(
    BTNode<Item_Type>* &local_root,
    const Item_Type& item) {
  if (local_root == NULL) {
    local_root = new BTNode<Item_Type>(item);
    return true;
  } else {
    if (item < local_root->data)
      return insert(local_root->left, item);
    else if (local_root->data < item)
      return insert(local_root->right, item);
    else
      return false;
  }
}
```

13

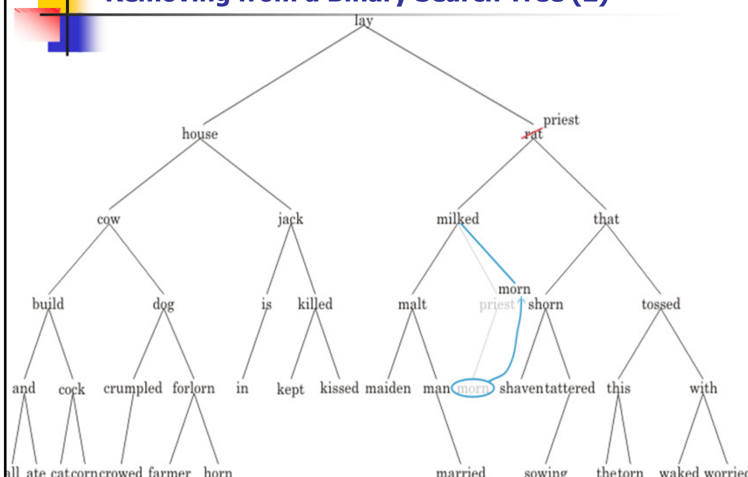
Removing from a Binary Search Tree

- Item ***not present:*** do nothing
- Item present in ***leaf:*** *remove leaf* (change to *null)*
- Item in *non-leaf* with *one child:*

  Replace current node *with that child*
- Item in *non-leaf* with *two children?*
  - Find *largest item* in the *left subtree*
  - *Recursively remove* it
  - Use it as the *parent* of the two subtrees
  - ***(Could use smallest item in right subtree)***

14

Removing from a Binary Search Tree (2)



15

The `erase` function

```
template<typename Item_Type>
  bool Binary_Search_Tree<Item_Type>::erase(
    const Item_Type& item) {
  return erase(this->root, item);
}

template<typename Item_Type>
  bool Binary_Search_Tree<Item_Type>::erase(
    BTNode<Item_Type>*  &local_root,
     const Item_Type& item) {
  if (local_root == NULL) {
    return false;
  } else {
    if (item < local_root->data)
      return erase(local_root->left, item);
    else if (local_root->data < item)
      return erase(local_root->right, item);
    else { // Found item
    . . .
```

16

The `erase` function (2)

```
BTNode<Item_Type>* old_root = local_root;
if (local_root->left == NULL) {
  local_root = local_root->right;
} else if (local_root->right == NULL) {
  local_root = local_root->left;
} else {
  replace_parent(old_root, old_root->left);
}
delete old_root;
return true;
 }
}
```

17

`replace_parent`

```
template<typename Item_Type>
void Binary_Search_Tree<Item_Type>::replace_parent(
      BTNode<Item_Type>*  &old_root,
      BTNode<Item_Type>*  &local_root) {
  if (local_root->right != NULL) {
    replace_parent(old_root, local_root->right);
  } else {
    old_root->data = local_root->data;
    old_root = local_root;
    local_root = local_root->left;
  }
}
```

18

Binary Search Tree Analysis

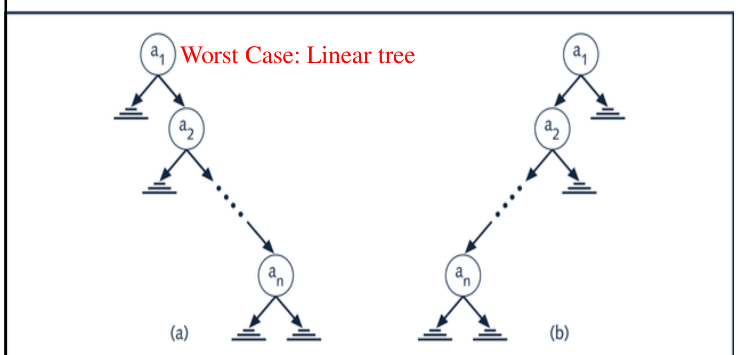


Figure 11-16 Linear binary search trees

19

Binary Search Tree Analysis

- **Theorem:** Let ***T*** be a binary search tree with ***n*** nodes, where ***n > 0***. The *average number of nodes visited* in a *search* of ***T*** is approximately ***$1.39\log_2 n$***
- *Number of comparisons* required to determine whether ***x*** is in ***T*** *(when successful)* is *one more* than the *number of comparisons* required to *insert **x** in **T***
- *Number of comparisons* required to *insert **x** in **T*** same as the number of comparisons made in *unsuccessful search*, reflecting that ***x** is not in **T***

20

Binary Search Tree Analysis

It follows that:

$$S(n) = 1 + \frac{U(0)+U(1)+\ldots+U(n-1)}{n} \qquad (11\text{-}1)$$

It is also known that:

$$S(n) = \left(1+\frac{1}{n}\right)U(n) - 3 \qquad (11\text{-}2)$$

Solving Equations (11-1) and (11-2)

$$U(n) \approx 2.77\log_2 n$$

and

$$S(n) \approx 1.39\log_2 n$$

21

Nonrecursive Inorder Traversal

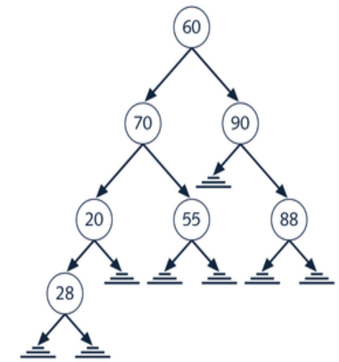


Figure 11-17 Binary tree; the leftmost node is 28

22

Nonrecursive Inorder Traversal: General Algorithm

```
1.  current = root;  //start traversing the binary tree at
                     // the root node
2.  while(current is not NULL or stack is nonempty)
     if(current is not NULL)
     {
        push current onto stack;
        current = current->llink;
     }
     else
     {
        pop stack into current;
        visit current;    //visit the node
        current = current->rlink;     //move to the
                                      //right child
     }
```

23

Nonrecursive Preorder Traversal: General Algorithm

```
1. current = root;  //start the traversal at the root node
2. while(current is not NULL or stack is nonempty)
    if(current is not NULL)
    {
       visit current;
       push current onto stack;
       current = current->llink;
    }
    else
    {
       pop stack into current;
       current = current->rlink;    //prepare to visit
                                    //the right subtree
    }
```

24

Nonrecursive Postorder Traversal

```
1. current = root;  //start traversal at root node
2. v = 0;
3. if(current is NULL)
      the binary tree is empty
4. if(current is not NULL)
   a. push current into stack;
   b. push 1 onto stack;
   c. current = current->llink;
   d. while(stack is not empty)
         if(current is not NULL and v is 0)
         {
            push current and 1 onto stack;
            current = current->llink;
         }
```


Nonrecursive Postorder Traversal (Continued)

```
else
 {
      pop stack into current and v;
      if(v == 1)
      {
            push current and 2 onto stack;
                  current = current->rlink;
                  v = 0;
      }
      else
                  visit current;
  }
```


Binary tree with visit functions

```
//Header File Binary Search Tree + NON recursive traversal + Visit Functions
#ifndef H_binaryTree
#define H_binaryTree

#include <iostream>

using namespace std;

   //Definition of the node
template<class elemType>
struct nodeType
{
  elemType                  info;
  nodeType<elemType>  *llink;
  nodeType<elemType>  *rlink;
};
```



```
 //Definition of the class
template <class elemType>
class binaryTreeType
{
public:
   const binaryTreeType<elemType>& operator=
            (const binaryTreeType<elemType>&);
   bool isEmpty();
   void inorderTraversal();
   void preorderTraversal();
   void postorderTraversal();

//        NEW Visit Functions

   void inorderTraversal(void (*visit) (elemType&));
     //Function to do an inorder traversal of the binary tree.
     //The parameter visit, which is a function, specifies
     //the action to be taken at each node.
```

```
 int treeHeight();
 int treeNodeCount();
 int treeLeavesCount();
 void destroyTree();
 binaryTreeType(const binaryTreeType<elemType>& otherTree);
 binaryTreeType();
 ~binaryTreeType();
//       New NON Recursive functions
 void nonRecursiveInTraversal();
 void nonRecursivePreTraversal();
 void nonRecursivePostTraversal();

protected:
 nodeType<elemType> *root;

private:
 void copyTree(nodeType<elemType>* &copiedTreeRoot,
            nodeType<elemType>* otherTreeRoot);
 void destroy(nodeType<elemType>* &p);
 void inorder(nodeType<elemType> *p);
```

29

Visit Functions

```
void inorder(nodeType<elemType> *p, void (*visit) (elemType&));
  //Function to do an inorder traversal of the binary
  //tree, starting at the node specified by the parameter p.
  //The parameter visit, which is a function, specifies the
  //action to be taken at each node.
```

30

```
 void preorder(nodeType<elemType> *p);
 void postorder(nodeType<elemType> *p);
 int height(nodeType<elemType> *p);
 int max(int x, int y);
 int nodeCount(nodeType<elemType> *p);
 int leavesCount(nodeType<elemType> *p);
};
```

31

Visit Functions

```
template <class elemType>
void binaryTreeType<elemType>::inorderTraversal
                (void (*visit) (elemType& item))
{
   inorder(root, *visit);
}

template <class elemType>
void binaryTreeType<elemType>::inorder(nodeType<elemType>* p,
                       void (*visit) (elemType& item))
{
   if(p != NULL)
   {
        inorder(p->llink, *visit);
        (*visit)(p->info);
        inorder(p->rlink, *visit);
   }
}
```

32

Binary Search Tree

```
//Header File Binary Search Tree

#ifndef H_binarySearchTree
#define H_binarySearchTree
#include <iostream>
#include <cassert>
#include "binaryTree.h"

using namespace std;

template<class elemType>
class bSearchTreeType: public binaryTreeType<elemType>
{
public:
    bool search(const elemType& searchItem);
      //Function to determine if searchItem is in the binary
      //search tree.
      //Postcondition: Returns true if searchItem is found in the
      //            binary search tree; otherwise, returns false.
```

33

```
    void insert(const elemType& insertItem);
      //Function to insert insertItem in the binary search tree.
      //Postcondition: If no node in the binary search tree has
      //          the same info as insertItem, a node with the
      //          info insertItem is created and inserted in the
      //binary search tree.

    void deleteNode(const elemType& deleteItem);
      //Function to delete deleteItem from the binary search tree
      //Postcondition: If a node with the same info as deleteItem
      //            is found, it is deleted from the binary
      //            search tree.

private:
    void deleteFromTree(nodeType<elemType>* &p);
      //Function to delete the node, to which p points, from the
      //binary search tree.
      //Postcondition: The node to which p points is deleted
      //            from the binary search tree.
};
```

34

```
    // implementation
template<class elemType>
bool bSearchTreeType<elemType>::search(const elemType&
    searchItem)
{   nodeType<elemType> *current;
    bool found = false;

    if(root == NULL)
      cerr<<"Cannot search the empty tree."<<endl;
    else
    { current = root;
      while(current != NULL && !found)
      {     if(current->info == searchItem)
              found = true;
            else
               if(current->info > searchItem)
                  current = current->llink;
               else
                  current = current->rlink;
      }  //end while
    }    //end else
    return found;
}        //end search
```

35

```
template<class elemType>
void bSearchTreeType<elemType>::insert(const elemType& insertItem)
{
    nodeType<elemType> *current;      //pointer to traverse the tree
    nodeType<elemType> *trailCurrent;  //pointer behind current
    nodeType<elemType> *newNode;      //pointer to create the node

    newNode = new nodeType<elemType>;
    assert(newNode != NULL);
    newNode->info = insertItem;
    newNode->llink = NULL;
    newNode->rlink = NULL;
```

```
if(root == NULL)
      root = newNode;
   else
   { current = root;
      while(current != NULL)
      {  trailCurrent = current;

          if(current->info == insertItem)
          { cerr<<"The insert item is already in the list -- ";
             cerr<<"duplicates are not allowed."<<endl;
             return;
          }
          else
             if(current->info > insertItem)
                current = current->llink;
             else
                current = current->rlink;
      }  //end while

      if(trailCurrent->info > insertItem)
         trailCurrent->llink = newNode;
      else
         trailCurrent->rlink = newNode;
   }
}        //end insert
```

37

```
template<class elemType>
void bSearchTreeType<elemType>::deleteNode(const elemType& deleteItem)
{  nodeType<elemType> *current;       //pointer to traverse the tree
   nodeType<elemType> *trailCurrent;        //pointer behind current
   bool found = false;

   if(root == NULL)
         cerr<<"Cannot delete from the empty tree."<<endl;
   else
   {     current = root;
         trailCurrent = root;

         while(current != NULL && !found)
         {        if(current->info == deleteItem)
                           found = true;
                  else
                  {        trailCurrent = current;
                           if(current->info > deleteItem)
                                    current = current->llink;
                           else
                                    current = current->rlink;
                  }
         }        //end while
```

```
      if(current == NULL)
            cout<<"The delete item is not in the tree."<<endl;
      else
            if(found)
            {
                  if(current == root)
                        deleteFromTree(root);
                  else
                        if(trailCurrent->info > deleteItem)
                              deleteFromTree(trailCurrent->llink);
                        else
                              deleteFromTree(trailCurrent->rlink);
            }     //end if
   }
}  //end deleteNode
```

39

```
template<class elemType>
void bSearchTreeType<elemType>::deleteFromTree
                              (nodeType<elemType>* &p)
{
    nodeType<elemType> *current;     //pointer to traverse the tree
    nodeType<elemType> *trailCurrent;        //pointer behind current
    nodeType<elemType> *temp;        //pointer to delete the node

    if(p == NULL)
       cerr<<"Error: The node to be deleted is NULL." <<endl;
    else if(p->llink == NULL && p->rlink == NULL)
        {  temp = p;
           p = NULL;
           delete temp;
        }
    else if(p->llink == NULL)
        {  temp = p;
           p = temp->rlink;
           delete temp;
        }
    else if(p->rlink == NULL)
        {  temp = p;
           p = temp->llink;
           delete temp;
        }
```

0

```
 else
    { current = p->llink;
      trailCurrent = NULL;

      while(current->rlink != NULL)
      {  trailCurrent = current;
         current = current->rlink;
      } //end while

      p->info = current->info;

      if(trailCurrent == NULL)    //current did not move;
                          //current == p->llink; adjust p
        p->llink = current->llink;
      else
        trailCurrent->rlink = current->llink;

      delete current;
    }    //end else
} //end deleteFromTree

#endif
```

41

Test for BST + Visit Functions - Client example

```
//   Test Binary Search Trees & Visit Functions
#include <iostream>
#include "binarySearchTree.h"
using namespace std;
void print(int& x);
void update(int& x);
int main()
{  bSearchTreeType<int>  treeRoot;                 //Line 1
   int num;                                        //Line 2
   cout<<"Line 3: Enter numbers ending with -999"
         <<endl;                                   //Line 3
   cin>>num;                                       //Line 4

   while(num != -999)                              //Line 5
   {     treeRoot.insert(num);                     //Line 6
         cin>>num;                                 //Line 7
   }
```

```
   cout<<endl<<"Line 8: Tree nodes in inorder: ";      //Line 8
   treeRoot.inorderTraversal(print);                   //Line 9
   cout<<endl<<"Line 10: Tree Height: "<<treeRoot.treeHeight()
   <<endl<<endl;                                       //Line 10
   cout<<"Line 11: ** Update Nodes **"<<endl;          //Line 11
   treeRoot.inorderTraversal(update);                  //Line 12

   cout<<"Line 13: Tree nodes in inorder after "
         <<"the update: "<<endl<<"        ";           //Line 13
   treeRoot.inorderTraversal(print);                   //Line 14
   cout<<endl<<"Line 15: Tree Height: "<<treeRoot.treeHeight()
              <<endl;                                  //Line 15
   return 0;                                           //Line 16
}
void print(int& x)                                     //Line 17
{
   cout<<x<<" ";                                       //Line 18
}
void update(int& x)                                    //Line 19
{
   x = 2 * x;                                          //Line 20
}
```