Data Structures

Binary Search Trees

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
4. http://inside.mines.edu/~dmehta/

Search structure

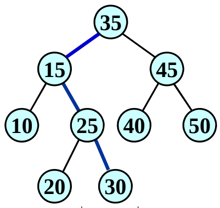
- The most common objective of computer is to store and retrieve data
- An efficient ways to organize collections of data records
  - Be stored and retrieved quickly
  - Such as **dictionary**
- Dictionary is a collection of record pairs <element, key>
  - Each pair has a key and an associated element
  - Assumption no two pair have the same key
- Dictionary provides operations for storing records, searching records and removing records from the collection

$_2$>,…,<e_n,k_n>

- Search for records might wish to search for the Key
  - Example 1 : given a particular key value *K* , find an element with key value k_j = *K*
  - Example 2 : find the fifth smallest element…
  - …
- Result of a search
  - Successful : is found the record pair with k_j in *D*
  - Unsuccessful : is not found or no such record pair exists in *D*

%LQDU\ 6HDUFK 7UHH %67

- Definition
  - A binary tree
  - Each node has a (key, value) pair
  - For every node x
    - all keys in the *left* subtree of x are smaller than that in x
    - all keys in the *right* subtree of x are greater than that in x



4

Class Definition

```
#include <iostream.h>
#include <stdlib.h>
template <class E,class K>
struct BSTNode
{                                          // ¼ ù A4 %é1«
   E data;                                      //   ž
   BSTNode<E,K> *left, *right;           //   € £ ¼ # € £

  //   ….

};
```

5

```
template <class E,class K>
class BST {
public:
   BST() { root = NULL; }                    ´EP -
   BST(K value);                   ´EP -
      BST() {};                             À ´ -
   bool Search (const K x) const
                  { return Search(x,root) != NULL; } // L2R
   BST<E>& operator = (const BST<E,K>& R); //FýD-U›B{ l
   void makeEmpty() { makeEmpty (root); root = NULL;} //4ž/ª
   void PrintTree() const { PrintTree (root); }          //DÃ *
   E Min() { return Min(root)  data; }        //!r 0 ?
    E Max() { return Max(root)  data; }      //!r 0 W
    bool Insert (const E & e1)
                          { return Insert(e1, root); } //   • à s2P
    bool Remove (const K x)
                          { return Remove(x, root);} // PK" [  x+´4 %é
```

6

```
private:
    BSTNode<E,K> *root;  // i 7I¸
    K RefValue;                             //DÃ • Œ ' 7
    BSTNode<E,K> *
        Search (const K x, BSTNode<E,K> *ptr);      //EB ,U› L2R

    void makeEmpty (BSTNode<E,K> *& ptr);      //EB ,U›4ž/ª
    void PrintTree (BSTNode<E,K> *ptr) const;     //EB ,U› ƒ
    BSTNode<E,K> *
        Copy (const BSTNode<E,K> *ptr);               //EB ,U› = f

    BSTNode<E,K>* Min (BSTNode<E,K>* ptr); //EB ,U›!r 0 ?
    BSTNode<E,K>* Max (BSTNode<E,K>* ptr); //EB ,U›!r 0 W

    bool Insert (const E& e1, BSTNode<E,K>*& ptr); //EB ,U›   •
    bool Remove (const K x, BSTNode<E,k>*& ptr); //EB ,U›  PK"
};
```



```
//RecursiveU›
// X    ptr j i+´ ¼ ù L2R A ] L2R [       x+´4 %é
//8  ® `U• I -  E   A 4 %é+´ ` pU• V I -  E                  NULL I
template<class E,class K>
BSTNode<E,K>* BST<E,K>::
          Search (const K x, BSTNode<E,K> *ptr)
{
    if (ptr == NULL) return NULL;
    else if (x < ptr->data) return Search(x, ptr->leftNULL*6 TDm038 Tc-.0163 Tw[(else )10.2(if (x <)6.1( )-10.2(pt)6.1(r->)6.2(da)8.9(t)6.1(a)-1.3())10.3( )20.4(ret)6.1(u)-1.3(rn )20.4(S)7.5(earch(x,)6.1( )
                                                                U›
                                                    //     ptr j i+´ ¼ ù L2R A ] L2R [       x+´4 %é
                                                   //8  ® `U• I -  E   A 4 %é+´ ` pU• V I -  E                  NULL
```

Insertion Operation

```
template <class E,class K>
bool BST<E,K>::Insert (const E& e1, BSTNode<E,K> *& ptr )
{  // .ñ 9 -  U›
   //  X    ptr j i+´ ¼ ù L2R A ]   • l j          e1+´4 %é
   // 8  X A ] " 9 [      e1+´4 %é, l =   •
       if (ptr == NULL) {          // à4 %é Œ j &4 %é   •
         ptr = new BstNode<E>(e1);       // K * à4 %é
         if (ptr == NULL)
            { cerr << "Out of space" << endl;  exit(1); }
          return true;
       }
      else if (e1 < ptr   data) Insert (e1, ptr   left); //   € A   •
      else if (e1 > ptr   data) Insert (e1, ptr   right); // # € A   •
      else return false;                        //x " X A ]    , = ½   •
   };
```



```
template <class E, class K>
BST<E,K>::BST (K value)
{
 //DÃ • 0 Z s2P ¿ G ,  */û 0 % ¼ ù L2R A
   E x;
   root = NULL;  RefValue = value;              //4ž/ª A
   cin >> x;                                   //DÃ •   ž
   while ( x.key != RefValue) {
                                   //RefValue _ 0 ZDÃ •4  • 7
       Insert (x, root);  cin >> x;         //   •   , ½DÃ •   ž
   }
}
```


Deletion Operation

- When remove a node from a BST
  - Deletion of a leaf
    - Its parent is set to 0, and the node disposed
  - Deletion of a nonleaf that has only one child
    - The node is disposed, and the single-child takes the place of the node
      - left child replace the disposed node
      - right child replace the disposed node
  - Deletion of a nonleaf that has two children
    - The node is replaced by either the largest node in its left subtree or the smallest one in its right subtree
    - Then the replacing node be proceed to remove from the subtree from which it was taken

Deletion Operation

```
// X    ptr  j i+´ ¼ ù L2R A ] PK" [       x +´4 %é
template <class E,class K >
bool BST<E,K>::Remove (const K x, BstNode<E,K> *& ptr)
{
   BstNode<E> *temp;
   if (ptr != NULL)
   {
      if (x < ptr   data) Remove (x, ptr   left);
                                    // X   € A ] —=| PK"
        else if (x > ptr   data) Remove (x, ptr   right);
                                          // X # € A ] —=| PK"
```

13

```
else if (ptr   left != NULL && ptr   right != NULL)
{                  //ptr 7.j £J^-1 j     x+´4 %éU• ³ 9 T Z € £
   temp = ptr   right;
                   // ` # € A L + ] ¿ ;0\ 0 Z4 %é
   while (temp   left != NULL)
        temp = temp   left;
   ptr   data = temp   data;
                  //*XA 4 %é   ž   / i4 %é   ž
     Remove (ptr   data, ptr   right);
}
```

14

```
else {          //ptr 7.j £J^-1 j     x+´4 %é 9 0 Z € £
         temp = ptr;
           if (ptr   left == NULL)  ptr = ptr   right;
         else ptr = ptr   left;
         delete temp;  // disposed
         return true;
      }
   }
   return false;
}
```

15

Data Structures

Thread Binary Trees

Teacher : Wang Wei

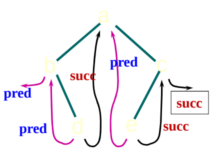
1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
4. http://inside.mines.edu/~dmehta/

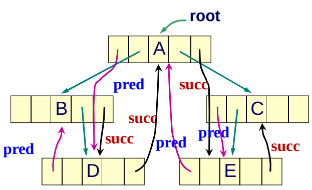
Threaded Binary Tree

- Using the threads and without an additional stack
  - Perform an traversal
  - Find the successor of any arbitrary node
  - Perform an traversal
  - Perform an traversal

Thread Binary Tree and Nodes

pred	leftChild	data	rightChild	succ





- predecessor thread pointer pre
- successor thread pointer succ

Nodes sturcture

- Distinguish between threads and normal pointer
  - Adding two Boolean fields : Ltag and Rtag
  - Let t be a pointer to a tree node
  - If tg , then tC contains a thread; otherwise contains a pointer to the left child
  - If tR , then tC contains a thread; otherwise contains a pointer to the right child

leftChild	Ltag	data	Rtag	rightChild

Inorder Thread Binary Tree

```
template <class T>
void ThreadTree<T>::createInorderThread()
{
    ThreadNode<T> *pre = NULL;            // }O¡4 %é 7I¸
     if (root != NULL) {                  //LŽ/ª ¼ ù A , 3ï2R F
        createInorderThread (root, pre);  // ] ¿E} ¶3ï2R F ¼ ù A
         pre   !rightChild = NULL;
        pre   !Rtag = 1;                  // > 4)6 ] ¿ 0 > 0 Z4 %é
     }
}
```



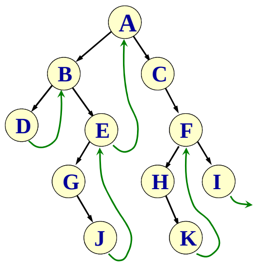
```
template <class T>
void ThreadTree<T>::createInThread(ThreadNode<T> *current,
                                   ThreadNode<T> *& pre)
{
    //EJD÷ ] ¿E} ¶ , ) ¼ ù AE =|3ï2R F
    if (current == NULL) return;
    createInThread (current->leftChild, pre);  //EB , ,    € A3ï2R F
    if (current   !leftChild == NULL)
   {
        // */û ƒ }4 %é+´ }O¡3ï2R
        current   !leftChild = pre;
        current   !Ltag = 1;
    }
```

```
    // */û }O¡4 %é+´ >4 3ï2R
    if (pre != NULL && pre   !rightChild == NULL)
    {    pre   !rightChild = current;
         pre   !Rtag = 1;
     }
     pre = current;      // }O¡C  :   , ƒ } 7I¸ A }E} ¶
     createInThread (current   !rightChild,  pre); //EB , ,  # € A3ï2R F
}
```

22

Finding the inorder successor of current Node

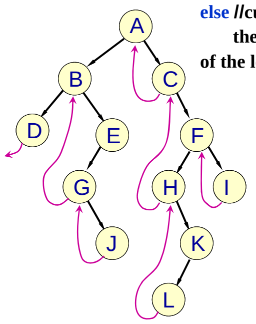
if (current !Rtag ==1) successor is current !rightChild
else //current !Rtag == 0
the inorder successor is the first node of the right subtree of current node



23

Finding the inorder predecessor of current Node

if (current !Ltag ==1)
successor is current !leftChild
else //current !Ltag == 0
the inorder predecessor is the last node of the left subtree of current node



24

A(B(E, F), C, D(G))field not shown

4. Degree-Two Representation

- Using binary tree
  - Rotate the right-sibling pointers in a left child-sibling tree clodkwise by 45 degrees

- Node structure

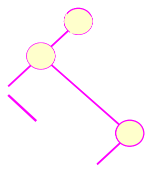
firstChild	data	nextSibling

Abstract Data Type of Tree

5. Parent Representation

- **One possible representation for sets**
  - Each set is represented as a tree
- **Linked the nodes from the**

	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	-1	0	0	0	1	1	3



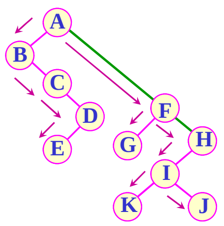
definition of a forest

- If F is a forest of trees, then the binary tree corresponding to this forset, denoted by

  F = {{T_1 = { r_1, T_{11}, …, T_{1k} }, T_2, ..., T_m}

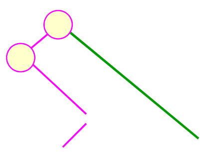
(1) is empty if n=0
(2) has root equal to root(T_1) r_1
(3) has left subtree equal to {T_{11}, …, T_{1k}} , where T_{11}, …, T_{1k} are the subtrees of root(T_1)
(4) has right subtree {T_2, ...,T_m}

- if F = Ø , then return
- else // in forest preorder
  - Visit the root r_1 of the first tree of T_1
  - Traverse the subtree of the first tree {T_{11}, …, T_{1k}}
  - Traverse the remaining trees of F {T_2, ...,T_m}



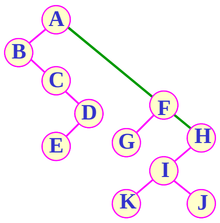
ABCDE FG HIKJ

- **if F = Ø , then return**
- **else** **// in forest inorder**
  - **Traverse the subtree of the first tree {T_{11}, …, T_{1k}}**
  - **Visit the root r_1 of the first tree of T_1**
  - **Traverse the remaining trees of F {T_2, ...,T_m}**



BCEDAGFKIJH

- if F = Ø , then return
- else // in forest inorder
  - Nodes are visited by level, beginning with the roots of each tree in the forest
  - Within each level, nodes are visited from left to right



AFH BCDGIJ EK

Data Structures

Union-Find Set

Teacher : Wang Wei

1. Ellis Horowitz,etc., Fundamentals of Data Structures in C++
2. ,
3. ,
4. http://inside.mines.edu/~dmehta/

47

Disjoint Sets

- Given a set {1, 2, …, n} of n elements
- Initially each element is in a different set
  - {1}, {2}, …, {n}
- Assume
  - The elements of the sets are the numbers *0,1,2,…,n-1*
  - The sets being represented are pairwise disjoint
- Example
  - S_1 = {0,6,7,8}
  - S_2 = {1,4,9}
  - S_3 = {2,3,5}

48

Initial a Union-Find Set (UFS)

- Each node is represented as a tree
- Using an array parent [] to represent the tree nodes
- parent[i] is the element that is the parent of element i

i	0	1	2	3	4	5	6	7	8	9
parent	1	1	1	1	1	1	1	1	1	1

- The root nodes *parent [i] = -1*

49

1

Constructor Function

```
// ´EP - U›
// sz  _Kö 8 s2P Z  U• ü â  3ô+´83 $ j      parent[0]-parent[size-1]

UFSets::UFSets(int sz)
{   size = sz;                          //Kö 8 s2P Z
    parent = new int[size];             // K * ü â  3ô
    for(int i = 0; i < size; i++)
        parent[i] = -1;                 // ÿ Z7  @ … s2PKö 8
};
```

52

Operations of UFS

- Operator
  - Union(root1, root2) //
    - Combines two sets into one
      - each of the n elements is in exactly one set at any time
  - Find(i) //
    - Identifies the set that contains a particular element i
  - UFSets(s) //

53

Strategy for *Find*

- Find(i)
  - start at the node that represents element i which given by parent[i]
  - follow parent fields until a root node whose parent field is null is reached
  - return element in this root node
  - Follow the tree, each node must have a parent pointer

```
int UFSets::Find(int i)
{   // Recursive Find,  L2R ¦E    5 [ s2P     x+´ A+´ i
    if (parent[i] < 0) return i;          // i+´  parent[] l ? ¾  0
    else return ( Find(parent[i]) );
};
//
int UFSets::Find(int i)  // Nonrecursive Find
{     while (parent[i] >= 0)
        i = parent[i];        // move up the tree
    return i;
}
```

54

Strategy for Union

- Union(i,j)
  - ƒ i and j are the roots of two different trees, i != j
  - to unite the trees, make one tree a subtree of the other
    - ƒ parent[j] = i

```
void UFSets::Union(int Root1, int Root2)
{  // Recursive Union, !r T Z =,( ÔKö 8   Root1 > Root2+´ ¦
    parent[Root1] += parent[Root2];
    parent[Root2] = Root1;        // 6 Root2E  Õ ` Root1 ;L'
};
```

55

Time Complexity

- The time taken a union operator is O(1)
- The n- 1 unions can be processed in time O(n)
- The time taken a find operator of the element i is O(i)
- The total time need to process the n finds is

$$O(\sum_{i=1}^{n} i) \quad O(n^2)$$

- and functions are very easy
- Their performance characteristics are not every good
  - Such as the degenerate tree (F0 F A)

56

Abstract Data Type of UFS

```
//Kö 8 ]+´ 4 Z €Kö 8 Â =,( Ô
const int DefaultSize = 10;
class UFSets
{
 public:
    UFSets (int sz = DefaultSize);              // ´EP -
        UFSets() { delete [] parent; }          // À ´ -
    UFSets& operator = (UFSets& R);             //Kö 8B{ l
    void Union (int Root1, int Root2);          // €Kö 8 ¦
    int Find (int x);                           //   ® x+´ i
 private:
    int *parent;         //Kö 8 s2P  3ô ( ü â=˜.j   )
    int size;            //Kö 8 s2P+´  ,
};
```

57