ECE 71/191T – Data Structures and Algorithms

Dr. Gregory R. Kriehn, Fresno State
C++ FINAL EXAM

Code, Write-Up, AND Demo Due By:
Thu, May 18, 8:45 – 10:45 AM

FINAL EXAM – B-Tree Implementation

Create a B-Tree by implementing the set.h class. Demonstrate that your B-Tree can insert and delete data within the tree, traverse the list, print the tree visually, etc. You may use the items.h template “helper” functions to help you implement the B-Tree.

Extra Credit: Implement the B-Tree using the STL vector class to hold the data and child pointers opposed to using arrays.

Extra Extra Credit: Modify the B-tree implementation of the set.h class so that each piece of data has a key (such as a string) and a data value (which may be any data type). When an item is inserted, both the key and value are specified. But in order to retrieve a value, all that is needed is the key. You may use the STL map member functions as a guide to create your own member function prototypes.

`set.h`

```
// CLASS IMPLEMENTED: Set
//   1. The items of the set are stored in B-tree,
//        satisfying the 6 rules.
//   2. The number of entries in the tree's root is in the
//        member variable data_count, and the number of subtrees
//        of the root is stored in the member variable
//        child_count.
//   3. The root's entries are store in data[0] through
//        data[data_count-1].
//   4. If the root has subtrees, then these subtrees are stored
//        in the sets *subset[0] through *subset[child_count-1].
//
// FOR REFERENCE, The 6 B-Tree Rules:
//   B-Tree Rule 1: Unless the whole set is empty, the root has
//        at least one entry; every other node has at least
//        MINIMUM entries.
//   B-Tree Rule 2: The maximum number of allowed entries in a
//        node is MAXIMUM (which is equal to 2*MINIMUM).
//   B-Tree Rule 3: The entries of each node are sorted from the
//        smallest entry (at data[0]) to the largest entry (at
//        data[data_count-1]).
//   B-Tree Rule 4: The number of subtrees below a non-leaf node
//        is always one more than the number of entries in the
//        node.
//   B-Tree Rule 5: For any non-leaf node: (a) data[i] is
//        greater than all the entries in subset[i], and (b)
//        data[i] is less than all the entries in subset[i+1].
//   B-Tree Rule 6: All leaves are at the same depth.


   Public:
     // CONSTRUCTORS AND DESTRUCTOR

     set();
      // Default Constructor
      // Postcondition:
      //  The set is empty

     set(const set &source);
      // Copy Constructor
      // Postcondition:
      //  The set is initialized with source

     ~set();
      // Destructor
      // Postcondition:
      //  The set is cleared


     // MODIFICATION MEMBER FUNCTIONS

     void operator=(const set &source);
      // Overload assignment operator
      // Postcondition:
      //  The set is assigned the value of source.

     void clear();
      // clear()
      // Postcondition:
      //  The set is empty.

     bool insert(const Item &entry);
      // insert()
      // Postcondition:
      //  If an equal entry was already in the set the set is
      //  unchanged and the return value is false. Otherwise,
      //  entry was added to the set and the return value is
      //  true/

     bool remove(const Item &target);
      // remove()
      // Postcondition:
      //  If target was in the set, then it has been removed
      //  from the set and the return values is true. Otherwise
      //  the set is unchanged and the return value is false.


     // CONSTANT MEMBER FUNCTIONS

     bool contains(const Item &target) const;
      // contains()
      // Postcondition:
      //  Returns a true/false value depending on whether target
      //  is found in the B-Tree.

     bool is_empty() const;
      // is_empty()
      // Postcondition:
      //  Returns true if the set is empty; otherwise returns
      //  false

     void print(int indent) const;
      // print()
      // Postcondition:
      //  Prints the B-Tree visually in tree format from left to
      //  right

   Private:
     // MEMBER CONSTANTS

     static const int MINIMUM = 1;           // Minimum Set size
     static const int MAXIMUM = 2*MINIMUM;   // Maximum Set size


     // MEMBER VARIABLES

     int data_count;          // Current num of items in data[]
     int child_count;         // Numb of children node points to
     Item data[MAXIMUM+1];    // Maximum data container size
     set *subset[MAXIMUM+2];  // Maximum children container size


     // MEMBER HELPER FUNCTIONS

     bool is_leaf() const;
      // is_leaf()
      // Postcondition:
        //  Returns true/false value depending on whether the node
        //  has any children

       bool loose_insert(const Item &entry);
        // loose_insert()
        // Precondition:
        //  The entire B-Tree is valid.
        // Postcondition:
        //  If entry was already in the set, then the set is
        //  unchanged. Otherwise, entry has been added to the set,
        //  and the entire B-tree is still valid EXCEPT that the
        //  number of entries in the root of this set may be one
        //  more than the allowed maximum.

       bool loose_remove(const Item &target);
        // loose_remove()
        // Precondition:
        //  The entire B-Tree is valid.
        // Postcondition:
        //  If target was in the set, then it has been removed
        //  from the set; otherwise the set is unchanged.  The
        //  entire B-tree is still valid EXCEPT that the number of
        //  entries in the root of this set may be one less than
        //  the allowed minimum.

       void remove_biggest(Item &removed_entry);
        // remove_biggest()
        // Precondition:
        //  (data_count > 0) and the entire B-Tree is valid.
        // Postcondition:
        //  The largest item in the set has been removed, and
        //  removed_entry has been set equal to a copy of this
        //  removed item. The B-Tree is still valid EXCEPT that
        //  the number of entries in the root of this set may be
        //  ones less than the allowed minimum.

       void fix_excess(int i);
        // fix_excess()
        // Precondition:
        //  (i < child_count) and the entire B-tree is valid
        //  EXCEPT that subset[i] has MAXIMUM+1 entries. Also, the
        //  root is allowed to have zero entries and one child.
        // Postcondtion:
        //  The tree has been rearranged so that the entire B-Tree
        //  is valid EXCEPT that the number of entries in the root
        //  of this set may be one more than the allowed maximum.
```

```
void fix_shortage(int i);
 // fix_shortage()
 // Precondition:
 //  (i < child_count) and the entire B-Tree is valid
 //  EXCEPT that subset[i] has only MINIMUM–1 entries.
 // Postcondition:
 //  the tree has been rearranged so that the entire B-Tree
 //  is valid EXCEPT that the number of entries in the root
 // of this set may be one less than the allowed minimum.

void transfer_left(int i);
 // transfer_left()
 // Precondition:
 //  (0<i<child_count) and (subset[i]->data_count>MINIMUM)
 //  and the entire B-tree is valid EXCEPT that
 //  subset[i-1] has only MINIMUM - 1 entries.
 // Postcondition:
 //  One entry has been shifted from the front of subset[i]
 //  up to data[i-1], and the original data[i-1] has been
 //  shifted down to the last entry of subset[i-1]. Also,
 //  if subset[i] is not a leaf, then its first subset has
 //  been transfered over to be the last subset of
 //  subset[i-1]. As a result, the entire B-tree is now
 //  valid.

void transfer_right(int i);
 // transfer_right()
 // Precondition:
 //  (i+1<child_count) and (subset[i]->data_count>MINIMUM)
 //  and the entire B-tree is valid EXCEPT that subset[i]
 //  has only MINIMUM - 1 entries.
 // Postcondition:
 //  One entry has been shifted from the end of subset[i]
 //  up to data[i], and the original data[i] has been
 //  shifted down to the first entry of subset[i+1]. Also,
 //  if subset[i] is not a leaf, then its last subset has
 //  been transfered over to be the first subset of
 //  subset[i+1]. As a result, the entire B-tree is now
 //  valid.

void merge_with_next_subset(int i);
 // merge_with_next_subset()

 // Precondition:
 //  (i+1<child_count) and the entire B-tree is valid
 //  EXCEPT that the total number of entries in subset[i]
 //  and subset[i+1] is 2*MINIMUM - 1.
 // Postcondition:
 //  subset[i] and subset[i+1] have been merged into one
 //  subset (now at subset[i]), and data[i] has been passed
 //  down to be the median entry of the new subset[i]. As a
 //  result, the entire B-tree is valid EXCEPT that the
 //  number of entries in the root of this set might be one
 //  less than the allowed minimum.
```