Some tvector functions

tvector( ); // default constructor (size==0)
tvector( int size ); // initial size of vector is size
tvector( int size, const itemType & fillValue ); // all entries == fillValue
tvector( const tvector & vec ); // copy constructor

int length( ) const; // support for old programs, deprecated
int capacity( ) const; // use in place of length()
int size( ) const; // # elements constructed/stored
void resize( int newSize ); // change size dynamically;
                                 // can result in losing values
void reserve(int size); // reserve size capacity
void push_back(const itemType& t);
void pop_back();
void clear(); // size == 0, capacity unchanged

Some definitions

struct Node {
    int value;
    Node * next;
    
    Node(int val, Node * nx)
        : value(val), next(nx) { }
};

struct TNode {
    int value;
    TNode * left;
    TNode * right;
    
    TNode(int val, TNode * lf, TNode * rt)
        : value(val), left(lf), right(rt) { }
};

struct GenNode {
    string name;
    GenNode * sibling;
    GenNode * child;
    
    GenNode(string & nm, GenNode * sib, GenNode * chd)
        : name(nm), sibling(sib), child(chd) { }
};
Algorithm for red-black tree insert

RB-Insert(root,x)
    Tree-Insert(root,x)  // binary search tree insert
    color x red
    while x≠root && x→p → color==red do
        if x→p is a left child then
            let y (a right child) denote the sibling of x→p
            if y is red then
                color x→p and y black       (case 1)
                color x→p→p red
                x = x→p→p
            else (y is black)
                if x is a right child then    (case 2)
                    x = x→p
                end if
                Left-rotate(root,x)
            end if
            color x→p black and x→p→p red    (case 3)
            Right-rotate(root,x→p→p)
        end if
    else (case if x→p is a right child)
        let y (a left child) denote the sibling of x→p
        if y is red then
            color x→p and y black        (case 1)
            color x→p→p red
            x = x→p→p
        else (y is black)
            if x is a left child then    (case 2)
                x = x→p
            end if
            Right-rotate(root,x)
        end if
        color x→p black and x→p→p red    (case 3)
        Left-rotate(root,x→p→p)
    end if
end while

color root black

PROBLEM 1 : (Sorting: 6 points)
Consider the following sorting algorithms: bubble sort, selection sort, insertion sort, quicksort, mergesort and radix sort.

Each group of numbers below represents one of these sorting algorithms. Each line represents a pass in that algorithm.

1. Which sorting algorithm above is the following sort?
2. Which sorting algorithm above is the following sort?

345 183 432 785 983 234 187 763
345 183 432 785 763 234 187 983
345 183 432 187 763 234 785 983
345 183 234 187 763 234 785 983
187 183 234 345 432 763 785 983
187 183 234 345 432 763 785 983
183 187 234 345 432 763 785 983

PROBLEM 2 :  (Stuck in the Middle: 12 points)
Consider the following problem: Given N elements in a data structure, find the middle element in sorted order. Note the elements are not necessarily in sorted order unless the data structure forces them to be.

For example, if the numbers are:

34 76 87 12 8 96 83

Then the middle element in sorted order is 76. If the elements had been sorted, 76 would be in the middle.

For each of the following possible data structures assume the N elements are already in the data structure. For each data structure, 1) describe briefly in one line an efficient algorithm to find the middle element in sorted order (do not write code), and 2) give the worst case running time of the algorithm.

1. an unsorted array
2. a sorted array - (here the elements are already sorted)
3. a sorted linked list - (here the elements are already sorted)
4. a min heap
5. a balanced binary search tree
6. a hash table of size M

PROBLEM 3 :  (Huff: 8 points)

Build an efficient Huffman tree using the algorithm we discussed in class for the following:
Show all steps in building the tree. The counts for each character have already been calculated for you. Also note the blank is represented by an underscore (_).

```
  t h e c a i n _
  4 3 2 1 2 1 1 4
```

PROBLEM 4 :  (Traversal: 6 points)

Consider the following tree.

![Tree Diagram]

Give the output when a postorder traversal is performed with the tree above.

Output:

If there are N nodes in the tree, what is the worst case running time of this traversal?

PROBLEM 5 :  (Red-black tree: 8 points)

Insert the following numbers (in the given order) into an empty red-black tree. Identify red nodes with R beside the node, black nodes with B beside the node. Show the resulting tree after each number is inserted (so show 8 trees). You do not need to show NULL nodes. The red-black insertion algorithm is on page 3.

The numbers to insert are: 34, 22, 10, 16, 13, 18, 19, 17

PROBLEM 6 :  (Hashing it: 8 points)
PART A (4 pts):
Complete the Insert function for hashing using the Separate Chaining (or Linked Method). Node is defined on page 2.

```c
void Insert(int key, tvector<Node *> & table)
// pre: assume room in table
// post: inserts key into table using separate chaining collision resolution
{
    int loc = h(key); // use hash function to get location
}
```

PART B (4 pts):
Complete the Insert function for hashing using the Linear Collision method for resolution.

```c
void Insert(int key, tvector<int> & table)
// pre: assume room in table
// post: inserts key into table using linear collision resolution method
{
    int loc = h(key); // use hash function to get location
}
```

PROBLEM 7: (Numbers in Trees: 14 points)

PART A (8 pts): Write the function `NumRightChildOnly` whose header is shown below. `NumRightChildOnly` returns the number of nodes in the tree that have only one child AND the one child is a right child.

For example, in the tree T below `NumRightChildOnly(T)` returns 2.

```
int NumRightChildOnly(TNode * T)
// post: returns the number of nodes in the tree that have only one child
// that is a right child
{

```

Assuming T is a balanced tree, give a recurrence relation for this function. Do not solve the recurrence.
PART B (6 pts):
Write the function `SumInsertRight` whose header is shown below. `SumInsertRight` assumes key is in the tree. It then inserts a new node as the right child of the node with key, containing the sum of values from the root to this node. If there is a previous right child of key, it becomes a right child of the inserted node.

For example, the figure on the right is the result after three calls to `SumInsertRight` to the tree T on the left. The calls are `SumInsertRight(8,0,T) SumInsertRight(26,0,T)` and `SumInsertRight(89,0,T).

```c
void SumInsertRight(int key, int sum, TNode * T)
// pre: key is in the tree
// post: a new node is inserted into the tree as the right child of key
// containing the sum of values from the root to this node.
// If there is a previous right child of key, it becomes a right child
// of the inserted node.
{
```

PROBLEM 8 : (Where’s that child?: 12 points)

PART A (6 pts):
Write the function `IsAChild` whose header is shown below. `IsAChild` returns true if name is a child of T, otherwise it returns false.

For example, in the figure below, `IsAChild(T,"Rhys")` returns true, `IsAChild(T,"Erik")` returns false, `IsAChild(T->child->sibling,"Isaac")` returns true.
bool IsAChild(GenNode * T, const string & name)
// post: returns true if name is a child of T, false otherwise
{

PART B (6 pts):
Write the function AddNewChildren whose header is shown below. AddNewChildren adds a child
to every node with the parameter name for those nodes that previously had children, but did not
have a child with “name”. The new child can be added into any position of the children (1st child,
last child, etc.).

For example, the figure shown below is the result of the call AddNewChildren(T,"Amanda") on the
tree on the previous page.

void AddNewChildren(GenNode * T, const string & name)
// post: every node with children now has a child with the name "name"
{
