Supernode Binary Search Trees

Haejae Jung and Sartaj Sahni

University of Florida, Gainesville, Fl. 32611
{hjjung, sahni}@cise.ufl.edu

January 23, 2001

1 Introduction

Binary search trees such as red-black trees, AVL trees, and splay trees have been developed for the representation of dictionaries[1] ([5], for example). Suitably modified versions of these data structures result in the efficient implementation of single-ended [3] and double-ended priority queues. The number of applications of binary search trees that permit fast search, insert, and delete capability is sufficiently large that a red-black tree implementation is provided in the C++ Standard Templates Library as well as in Java's `java.util` package (the class `java.util.TreeMap` employs a red-black tree).

As far as we are aware, all implementations and discussions of binary search trees (for example [4, 2, 5, 6, 7] as well as the C++ STL and Java implementations of red-black trees) use a representation in which each node has exactly one pair/element of the dictionary. We shall refer to single element per node binary search trees as $BST(1)$ (unbalanced binary search trees with one element per node), $AVL(1)$, $RB(1)$, and $Splay(1)$.

In this paper, we propose supernode versions of unbalanced binary search trees as well as of red-black, AVL, and splay trees. We refer to these supernode varieties as $BST(m)$, $AVL(m)$, $RB(m)$, and $Splay(m)$. In a supernode BST ($BST(m)$, $m > 1$) each node may have up to m elements. More precisely, a $BST(m)$ satisfies the following properties:

1. **Binary Tree Property**
   A $BST(m)$ is a binary tree.

2. **Search Tree Property**
   The elements in each node x of the binary tree are such that the left subtree of x has no element that is larger than any element in x and the right subtree has no element that is smaller than any element in x.

[1] A *dictionary* is a collection of pairs of the form `(key, value)`. We assume that no two pairs in the dictionary have the same key. This assumption that the keys be distinct may be relaxed if necessary. The operations that may be performed on a dictionary are: search for a pair given its `key`; insert a pair; and delete a pair given its `key`.

3. **Node Occupancy Property**

   (a) Every nonleaf node has exactly m elements. In other words, nonleaf nodes are always full.

   (b) Leaf nodes have between 1 and m elements.

Figure 1 shows a possible node structure for a (super) node in a $BST(m)$. Each node has a left child, a right child and an optional parent pointer. Each node has the abilty to store m elements, where each element is a `(key, value)` pair. The node elements with the smallest key (min element) and the largest key (max element) are easily accessible. The remaining elements may or may not be ordered by key (the analyses and experiments reported in this paper assume that the remaining elements are not ordered).

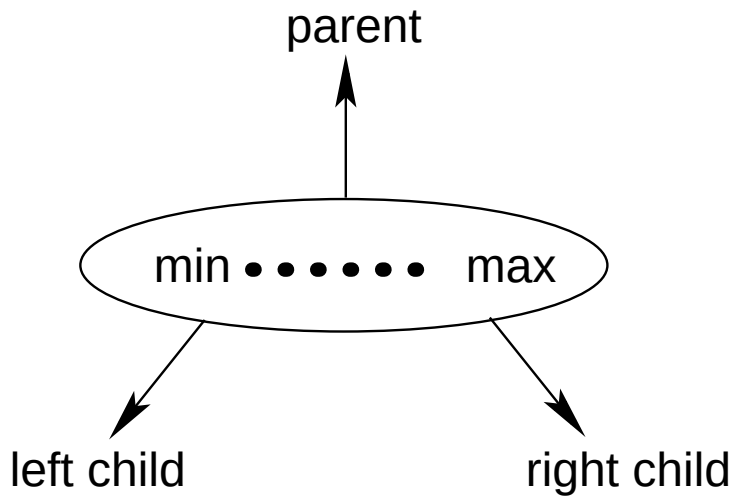


Figure 1: A supernode

Figure 2 gives the fields of a supernode. The field labeled *size* gives the number of pairs currently in the node (including the pairs with min and max keys).

Although m-way search trees store up to $m-1$ elements in a single node, m-way search trees are not binary trees when $m > 2$. Furthermore, m-way search trees (including their balanced varieties–B-trees) have a worst-case space requirement that exceeds that of binary search trees. This is a consequence of the fact that practical implementations of m-way search trees use fixed-size nodes (i.e., nodes that can accomodate $m-1$ elements along with pointers and other needed information) rather than variable-sized nodes (i.e., nodes that can accomodate only as many elements as are presently in the node plus associated information). Fixed-size nodes are preferred because the use of variable-sized nodes results in excessive and expensive dynamic storage allocation and deallocation.

Experimental results obtained by us indicate that the balanced supernode varieties of binary search trees considered by us provide both time and memory advantages relative to their respective single node per element versions. We did not experiment with unbalanced supernode binary search trees.

In Section 2 we describe how search, insert, and delete operations are performed in a $BST(m)$. Subsequent sections describe how these operations are done in $AVL(m)$, RB(m), and Splay(m) trees. Our experimental results are provided in Section 7.

Parent
minKey, minValue
maxKey, maxValue
size
key1, value1
key2, value2
•
•
•
LeftChild
RightChild

Figure 2: Fields in a supernode

2 $BST(m)$

2.1 Search

Algorithm 2.1 describes how a $BST(m)$ may be searched for a pair with key equal to `theKey`. The search begins at the root of the tree. If the search key is less (greater) than the min (max) key in the current node, the search moves into the left (right) subtree. If the search key lies between the min and max keys in the current node, the current node is examined to determine whether the node does, in fact, contain a pair with key equal to the search key. When the pairs in a node are not sorted by key, this determination takes $O(m)$ time. So, the complexity of our search algorithm is $O(h + m)$, where h is the height of the $BST(m)$.

2.2 Insert

Algorithm 2.2 describes how to insert a pair into a $BST(m)$. First we invoke the search method to determine whether the $BST(m)$ already has a pair that has the same key as the pair that is to be inserted. If so, the insert fails. If not, the insertion of the new pair is handled as several cases. To arrive at these cases, one must keep in mind that all nonleaf nodes of a $BST(m)$ are always full (i.e., they always have m pairs). When `currentNode` is `NULL` and has a parent that is not full, the new pair is inserted into this parent node. When `currentNode` is `NULL` and has no parent (here, by parent we mean the value `currentNode` had in Algorithm 2.1 just prior to the returned value) or has a parent that is full, the new pair is inserted into a new node that becomes the child of the full parent, or becomes the root (in case there is no parent).

When the current node is not `NULL`, the new pair has to be inserted into the current node. This is easy to do when the current node is not full. When the current node is full, the insert process depends on the degree of the current node. When this degree is 1 and the current node has no left child, the pair

Algorithm 2.1 $BST(m)$ search algorithm

```
// theKey: key value to find
Search(theKey)
{
    currentNode = root;
    while (currentNode != NULL)
        if (theKey < currentNode→minKey)
            currentNode = currentNode→leftChild;
        else if (theKey > currentNode→maxKey)
                currentNode = currentNode→rightChild;
             else break;

    if (currentNode == NULL) return <NULL, NOT_FOUND>;
    if (theKey is in currentNode) return <currentNode, FOUND>;
    return <currentNode, NOT_FOUND>;
}
```

with min key in the current node is replaced by the new pair, a new node is created for the removed min pair, and the new node becomes the left child of the current node. The case when the degree is 1 and the current node has no right child is similar. When the degree of the current node is 2, the pair with the min key in the current node is replaced by the new pair, and the removed min key pair is inserted into the the left subtree of the current node. The complexity of the insert method is easily seen to be $O(h + m)$.

2.3 Delete

Algorithm 2.3 gives the $BST(m)$ deletion algorithm. When removing a pair from a nonleaf node of a $BST(m)$, we must replace the removed pair with either the pair in the left subtree that has maximum key or the pair in the right subtree that has minimum key. This replacement process is continued until the replacing pair comes from a leaf node.

The complexity of our algorithm to delete a pair is $O(hm)$, because whenever a pair is moved from `largestNode` (`smallestNode`) to `currentNode`, we must determine the new max (min) pair in `largestNode` (`smallestNode`).

2.4 $BST(m)$ Properties

First, we bound the height of a $BST(m)$. For our definition of height, we assume that the height of an empty tree is 0, and that of a tree that has just a root node is 1.

Algorithm 2.2 $BST(m)$ insert algorithm

```
// (theKey, theValue): pair to insert
Insert(theKey, theValue)
{
    <currentNode, found> = Search(theKey);
    if (found) return FOUND;
    parentNode = currentNode→parent;
    if (currentNode == NULL)
        if (parentNode is neither NULL nor FULL)
            insert (theKey, theValue) into parentNode;
        else {create a node newNode with (theKey, theValue);
                attach newNode to parentNode (unless parent is NULL);}
    else if (currentNode is NOT FULL)
            insert (theKey, theValue) into currentNode;
    else if (currentNode has no left child) {
            take min data out of currentNode;
            put (theKey, theValue) into currentNode;
            create a node newNode with min data that was removed from currentNode;
            attach newNode to currentNode as its left child;}
    else if (currentNode has no right child) {
            take max data out of currentNode;
            put (theKey, theValue) into currentNode;
            create a node newNode with max data that was removed from currentNode;
            attach newNode to currentNode as its right child;}
    else {
            take min data out of currentNode;
            put (theKey, theValue) into currentNode;
            let largestNode be the node in the left subtree
                 of currentNode with largest key;
            if (largestNode is full) {
                create a node newNode with min data that was removed from currentNode;
                attach newNode to currentNode as its left child;}
            else
                put min data that was removed from currentNode into largestNode;}
    return INSERTED;
}
```

Algorithm 2.3 $BST(m)$ delete algorithm

```
// theKey: key of element to delete
Delete(theKey)
{
    <currentNode, found> = Search(theKey);
    if (not found) return NOT_FOUND;

    if (currentNode has a left child) {
        while (currentNode is not a leaf node) {
            let largestNode be the node in the left subtree
                 of currentNode with largest key;
            move the pair with largest key from largestNode to currentNode;
            currentNode = largestNode;}
       delete the pair with largest key from currentNode;}
    else if (currentNode has a right child) {
        while (currentNode is not a leaf node) {
            let smallestNode be the node in the right subtree
                of currentNode with smallest key;
            move the pair with smallest key from smallestNode to currentNode;
            currentNode = smallestNode;}
       delete the pair with smallest key from currentNode;}
    else delete the pair with key equal to theKey from currentNode;
    if (currentNode is empty) remove currentNode from tree;
    return DELETED;
}
```

Theorem 2.1 $\log_2(n/m+1) \leq height(BST(m)) \leq (n-1)/m+1$, *where* n *is the number of pairs in the* $BST(m)$.

Proof The lower bound follows from the lower bound of $\log_2(q+1)$ on the height of a q-node binary tree and the observation that an n-pair $BST(m)$ has at least n/m nodes.

For the upper bound, we note that a $BST(m)$ whose height is h has at least $h-1$ nonleaf nodes and at least 1 leaf node. Therefore, the number of pairs in the nonleaf nodes is at least $(h-1)m$ and the number in the leaf node is at least 1. So, $n \geq (h-1)m+1$. Therefore, $h \leq (n-1)/m+1$. ■

For comparison purposes, we note that the height of an n-pair binary search tree lies between $\log_2(n+1)$ and n. Although the use of supernodes reduces the best-case height by only about $\log_2 m$, the use of supernodes reduces the worst-case height by a factor of about m.

Next, we bound the space requirements of a $BST(m)$. For this analysis, we assume that the optional `parent` pointer is not implemented (the search, insert and delete operations we are considering do not require such a pointer), and that each pointer (left child, right child, key, value) takes 4 bytes. Further, each node has a 4 byte integer field `size`, which gives the number of pairs currently in the node (`size` equals m when the node is full). Experimental results (Section 7) indicate that the optimal value of m is in the tens. So, we can get by with a `size` field that is just 1 byte. However, since most real-world memory allocators allocate memory in multiples of 4 bytes, our assumption of a 4-byte `size` field reflects the real amount of memory that is allocated in practice. Under these assumptions, the space required by a single node of a $BST(m)$ is $8m+12$ bytes. By comparison, the space required by a node of a $BST(1)$ (such a node has left and right child pointers, and pointers to the key and value) is 16 bytes.

Theorem 2.2 *A* $BST(m)$ *with* n*-pairs requires at least* $(n/m)(8m+12)$ *bytes, and at most approximately* $16(m+n)$ *bytes, when* $m >> 1$.

Proof The minimum number of nodes in an n-pair $BST(m)$ is n/m. So, the minimum space requirement is $(n/m)(8m+12)$ bytes.

For the maximum space requirement, let T be a q-leaf $BST(m)$ that has the fewest number of pairs. Since T is a binary tree, T has at least $q-1$ nonleaf nodes (a q-leaf binary tree has exactly $q-1$ nodes whose degree is 2 and zero or more nodes whose degree is 1, Lemma 5.3 of [2]). Further, since T is a q-leaf $BST(m)$ that has the fewest number of pairs and since nonleaf nodes must have m pairs while leaves may have only one pair, we may assume that the number of nonleaf nodes is actually $q-1$. Therefore, the number of pairs n in T is given by $n=(q-1)m+q$ (each nonleaf has m pairs and each leaf has at least one pair). So, $q=(m+n)/(m+1)$. Since the total number of nodes in T is $2q-1$ (q leaf and $q-1$ nonleaf nodes), the space $S(T)$ required by T is

$$\begin{aligned} S(T) &= (2q-1)(8m+12) \\ &= 8(2\frac{m+n}{m+1}-1)(m+1.5) \\ &\approx 16(m+n) \end{aligned} \tag{1}$$

■

The worst-case *space ratio*, $SR(m,n)$, of an n-pair $BST(m)$ is the ratio of the the space taken by an n-pair $BST(1)$ and the maximum space taken by an n-pair $BST(m)$. From Theorem 2.2, we see that

$$\begin{aligned} SR(m,n) &\approx \frac{16n}{16(m+n)} \\ &= \frac{n}{m+n} \end{aligned} \tag{2}$$

From Equation 2, we see that a $BST(m)$ may take slightly more space than taken by a $BST(1)$. Comparing best cases for $BST(m)$s and $BST(1)$s gives a space ratio of $16n/((n/m)(8m+12)) = 4m/(2m+3) \approx 2$.

3 $AVL(m)$

3.1 Definition

Let T be a $BST(m)$. Let T' be the binary tree obtained when all nonfull nodes (note that only leaves may be nonfull) of T are removed from T. T' is called the *residual full node binary tree* (or simply *residual tree*, RT) corresponding to T. T' is denoted by $RT(T)$. T is an $AVL(m)$ iff $RT(T)$ is an AVL tree.

Balance factors are defined only for full nodes of T (i.e., for nodes of T that are also nodes of $RT(T)$). These balance factors are computed relative to $RT(T)$ (i.e., compute the balance factors in $RT(T)$ in the normal fashion for $RT(T)$, and then use these balance factors as the balance factors for the full nodes of T). The balance factor of a node x of $RT(T)$ is $height(x_L) - height(x_R)$, where x_L and x_R are, respectively, the left and right subtrees of x.

3.2 Operations

The search algorithm for an $AVL(m)$ is identical to that for a $BST(m)$.

To insert a new pair into an $AVL(m)$, first insert the pair using the algorithm (Algorithm 2.2) for $BST(m)$. If this insertion results in a previously nonfull node (must be a leaf) becoming full, then there is a possibility that the residual tree is not an AVL tree. The balance factor of the newly full node is set to 0, and we trace the path from this newly full node towards the root, adjusting balance factors as necessary. If a node whose adjusted balance factor is either $+2$ or -2 is encountered, one of the standard AVL rotations is performed with respect to this node and we discontinue tracing the path to the root as no additional balance factor adjustments will take place.

We need to verify that the standard AVL tree rotations performed as described above do not result in nonfull nodes that are not leaves. There are four rotation types used in an AVL tree: LL, LR, RL, RR. Since rotation types RR and RL are symmetric to LL and LR, respectively, we consider LL and LR rotations only.

Figure 3 shows an LL rotation. z is the node whose balance factor has become 2. In the case of an LL rotation with respect to z, the newly full node is in the left subtree of the left child y of z. Since

the tree is a $BST(m)$ (not necessarily an $AVL(m)$) immediately following insertion and just prior to the rotation, nodes x, y, and z are full. Therefore, following the LL rotation no nonfull leaf becomes a nonleaf (note that the subtrees a, b, c, and d may be empty). Therefore, following the rotation we have a $BST(m)$. Since the rotation ensures that the residual tree is an AVL tree, the tree following the insert-rebalance step is an $AVL(m)$.

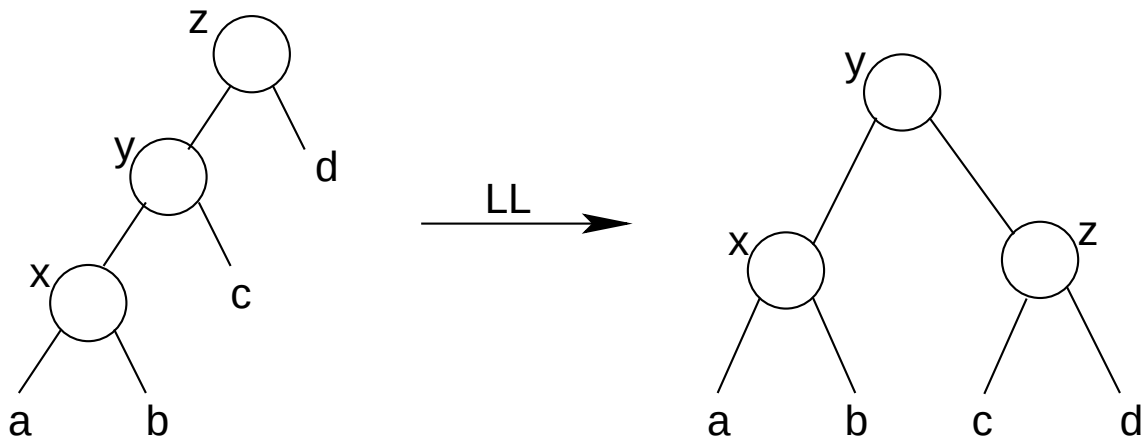


Figure 3: LL rotation

Figure 4 shows an LR rotation. Once again, z denotes the node whose balance factor has become 2. This time the newly full node is in the right subtree of the left child y of z. Since x, y, and z are full nodes, the rotation does not result in a nonfull nonleaf. Consequently, the tree that results from the rotation is an $AVL(m)$.

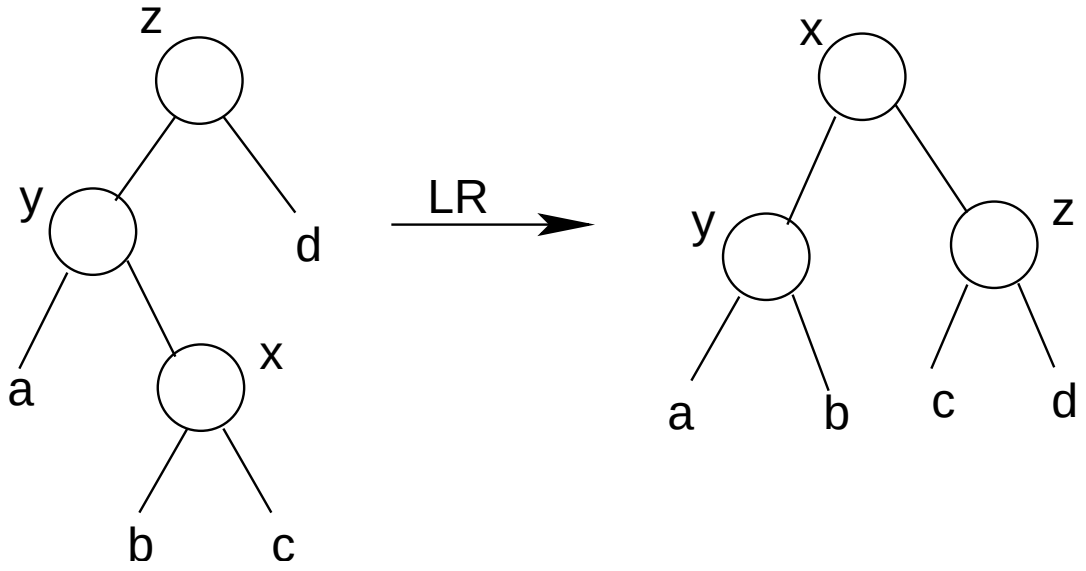


Figure 4: LR rotation

A pair may be removed from an $AVL(m)$ using the removal algorithm (Algorithm 2.3) for a $BST(m)$. If this removal results in no previously full node becoming nonfull, the residual tree is unchanged, and so

no rebalancing and/or balance factor adjustment steps are done. When a previously full node becomes nonfull, it is easy to see that there is exactly one node that was previously full and that is now nonfull, and that this newly nonfull node is a leaf. The residual tree changes and we begin a rebalancing and balance factor adjustment pass in which we follow the path from the parent (if any) of the newly nonfull node towards the root. It is easy to verify that rebalancing rotations (if any) performed along this path do not change leaf nodes to nonleaf nodes. So, following the rebalancing, every nonleaf node is full. Therefore, the rebalancing leaves behind an $AVL(m)$.

3.3 Properties

Theorem 3.1 *Let T be an $AVL(m)$ that has n-pairs, and let $height(T)$ be the height of T.*

$$\log_2(n/m+1) \leq height(T) \leq 1.44\log_2(n/m+2)+1$$

Proof The number of supernodes in T is at least n/m. Since, T is a binary tree that has at least n/m nodes, its height is at least $\log_2(n/m+1)$.

For the upper bound on the height of T, consider the residual tree $RT(T)$. From the definition of $AVL(m)$, it follow that $RT(T)$ is an AVL tree. From the well known height bounds on AVL trees (see [4, 2, 5]), for example), it follows that

$$\log_2(q+1) \leq height(RT(T)) \leq 1.44\log_2(q+2) \tag{3}$$

where q is the number of nodes in $RT(T)$. Since every node of $RT(T)$ is a full node, there are $qm \leq n$ pairs in $RT(T)$. Therefore, $q \leq n/m$. From this, Equation 3, and the fact that $height(RT(T)) \leq height(T) \leq height(RT(T))+1$, we obtain

$$height(T) \leq 1.44\log_2(n/m+2)+1 \tag{4}$$

■

Notice that even though the worst-case height of a $BST(m)$ is about $1/m$ that of a $BST(1)$ that has the same number of pairs, the worst-case height of an $AVL(m)$ is only about $1.44\log_2(m)-1$ less than that of an $AVL(1)$ that has the same number of elements. So, for example, when we use an $AVL(16)$ over an $AVL(1)$, the worst-case height is reduced by about 4 only (independent of the number of pairs).

Theorem 3.2 *The complexity of the search, insert, and delete operations of an n-pair $AVL(m)$ are, respectively $O(\log n+m)$, $O(\log n+m)$, and $O(m\log n)$.*

Proof Follows from the observation that these operations take $O(h+m)$, $O(h+m)$, and $O(hm)$ time, respectively, where h is the tree height and from Theorem 3.1, which states that $height(AVL(m)) = O(\log n)$. ■

For the space requirements of an n-pair $AVL(m)$, we note that each supernode of an $AVL(m)$ must have a balance factor field and a parent field in addition to the fields shown in Figure 2 for the supernodes

of a $BST(m)$. Note that 2 bits suffice for the balance factor field. As noted in Section 2, the `size` field needs only 1 of the 4 bytes allocated to it. Therefore, when allocating memory in 4-byte multiples, the size of an $AVL(m)$ supernode is 4 bytes more than that of a $BST(m)$ supernode. So, the size of an $AVL(m)$ node is $8m + 16$ bytes. However, the size of an $AVL(1)$ node is 8 bytes more than that of a $BST(1)$ node, because there is no unused space in the 16 byte space requirement of a $BST(1)$ node to accomodate the balance factor field. So, the size of an $AVL(1)$ node is 24 bytes.

Theorem 3.3 *An $AVL(m)$ with n-pairs requires at least $(n/m)(8m + 16)$ bytes, and at most approximately $16(m + n)$ bytes, when $m >> 1$.*

Proof Same as that for Theorem 3.3. ■

From Theorem 3.3, it follows that the worst-case space ratio, $SR(m, n)$, of an n-pair $AVL(m)$ is

$$\begin{aligned} SR(m,n) &\approx \frac{24n}{16(m+n)} \\ &= \frac{3n}{2(m+n)} \end{aligned} \tag{5}$$

From Equation 5, we see that when $m < 0.5n$, an n-pair $AVL(m)$ takes less space than does an n-pair $AVL(1)$. For $n >> m >> 1$, $AVL(m)$s provide a worst-case space ratio of 1.5. Comparing best cases for $AVL(m)$s and $AVL(1)$s gives a space ratio of $24n/((n/m)(8m + 16)) = 3m/(m + 2) \approx 3$.

4 $RB(m)$

4.1 Definition

T is a $RB(m)$ iff the residual tree $RT(T)$ is a red-black tree. Node (or edge) colors are defined only for full nodes of T. These colors are identical to the colors assigned to these nodes in $RT(T)$.

4.2 Operations

The search algorithm for a $RB(m)$ is identical to that for a $BST(m)$.

For the insert and delete algorithms, we use the bottom-up variety of red-black trees for $RB(1)$. For $RB(m)$ insert and delete is done just as in $AVL(m)$ with the exception that the bottom to top rebalancing pass uses color flips and/or rotations to ensure that the residual tree following the operation also is a red-black tree.

More specifically. to insert a new pair, first insert the pair using the algorithm (Algorithm 2.2) for $BST(m)$. If this insertion results in a previously nonfull node (must be a leaf) becoming full, color the newly full node red and trace the path from this newly full node towards the root, flipping colors and/or performing rotations as necessary. We may verify that this color flipping/rotation pass does not result in nonfull nodes that are not leaves (this actually follows from our corresponding discussion for $AVL(m)$,

because the red-black rotations are the same as the AVL rotations and because color flips do not move nodes).

A pair may be removed from a $RB(m)$ using the removal algorithm (Algorithm 2.3) for a $BST(m)$. If a previously full node becomes nonfull, it is easy to see that there is exactly one node that was previously full and that is now nonfull, and that this newly nonfull node is a leaf. The residual tree changes and we begin a rebalancing pass in which we follow the path from the parent of the newly nonfull node towards the root. It is easy to verify that rebalancing rotations and color flips (if any) performed along this path do not change leaf nodes to nonleaf nodes. So, following the rebalancing, every nonleaf node is full. Therefore, the rebalancing leaves behind a $RB(m)$.

4.3 Properties

Theorem 4.1 *Let T be a $RB(m)$ that has n-pairs, and let $height(T)$ be the height of T.*

$$\log_2(n/m+1) \leq height(T) \leq 2\log_2(n/m+1)+1$$

Proof The proof is very similar to that of Theorem 3.1 and uses the fact that the height of an q-node red-black tree is at most $2\log_2(q+1)$ (see [2, 5, 6]), for example). ■

The asymptotic time complexity of $RB(m)$ operations is the same as that for $AVL(m)$ operations and the best- and worst-case space requirements are also the same.

5 $Splay(m)$

5.1 Bottom-Up Splay Trees

We consider two versions of $Splay(m)$–bottom up and top down. These are modeled after the bottom up and top down single element per node splay trees of Sleator and Tarjan [7]. A bottom up $Splay(m)$ ($buSplay(m)$) is a $BST(m)$ in which each operation (search, insert, and delete) is done as in a $BST(m)$ and in which each operation is followed by a splay that begins at a designated node (called the splay node). The splay uses rotations to successively move the splay node up the tree two levels at a time, possibly culminating with a one-level move. Following the splay, the splay node is the root of the $buSplay(m)$.

For each of the operations (search, insert, delete), the splay node is the last (i.e., deepest) full node (if any) encountered by the corresponding $BST(m)$ algorithm. The full/nonfull status of a node is with respect to the tree following the execution of the $BST(m)$ algorithm. Note that in the case of an insert, the splay node may be a newly full node, and in the case of a delete, the splay node may be the parent of a node that was full prior to the delete but is not full after the delete (i.e., a newly nonfull node).

There are 6 types of rotations that are used in a splay–L, R, LL, RR, LR, and RL. Since the R, RR, and RL rotation types are symmetric to the remaining 3, we consider only the L, LL, and LR types here. Figures 5, 6, and 7 show these three rotation types. In each figure, x is the splay node. A quick examination of these figures reveals that none of the rotations results in a nonfull node that is not a leaf. Note that the splay node x is always a full node.

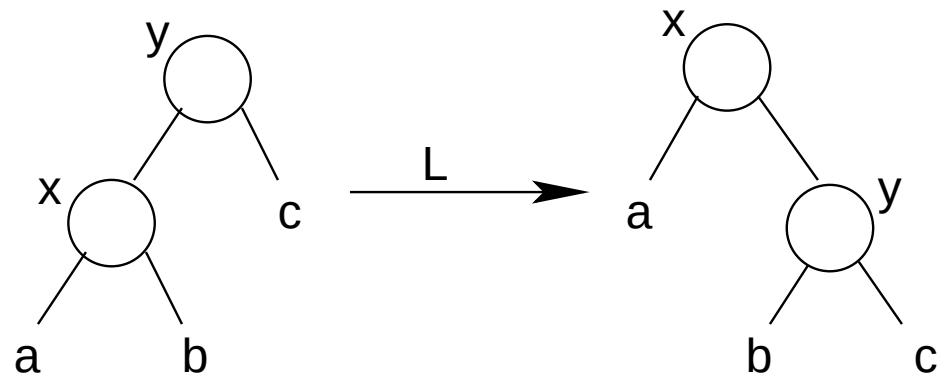


Figure 5: L rotation for bottom-up splay

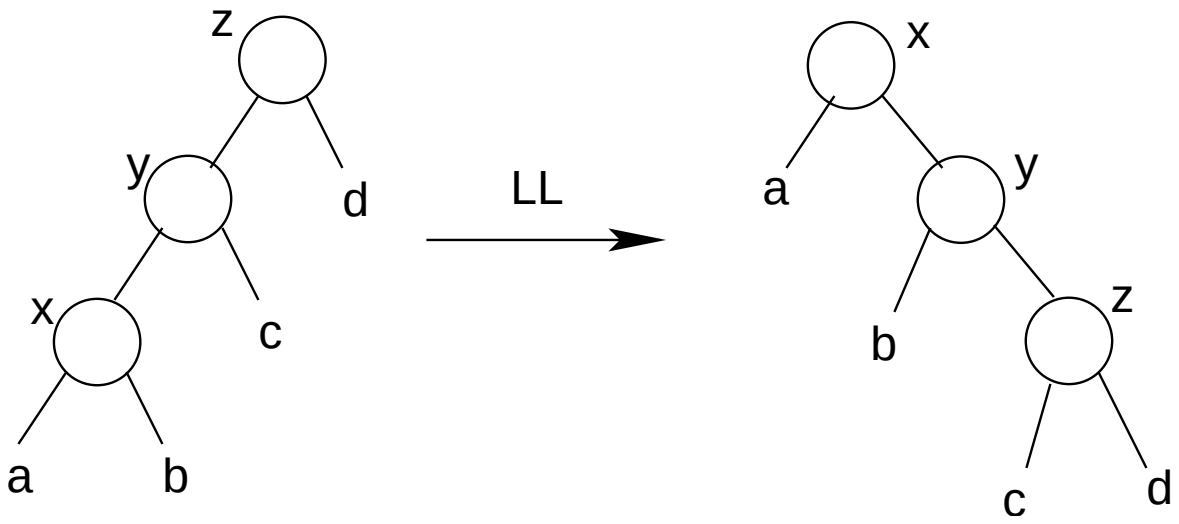


Figure 6: LL rotation for bottom-up splay

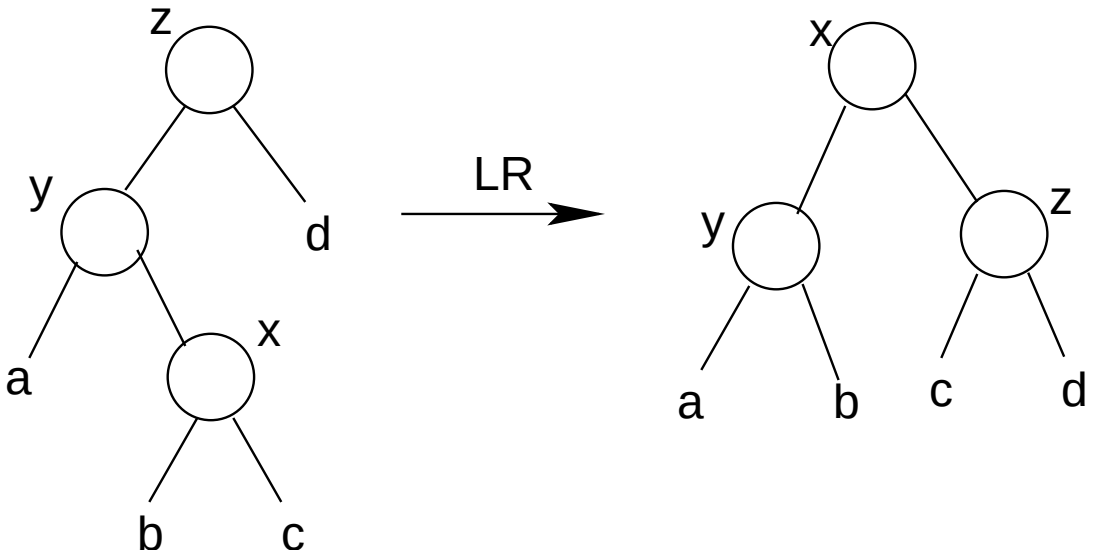


Figure 7: LR rotation for bottom-up splay

5.2 Top-Down Splay Trees

A top down $Splay(m)$ ($tdSplay(m)$) is a $BST(m)$ in which each operation (search, insert, and delete) is done as in a $BST(m)$. However, while the operation is being performed we maintain three trees–$Small$, $Middle$, and Big. Initially $Small$ and Big are empty, and $Middle$ is the $tdSplay(m)$ that we start with. When moving down the $Middle$ tree to perform the desired operation we move two-levels down at a time (so long as this is possible). The last move down may be a one-level move.

Each time we move down two levels (or one in case of a final move), a splay step as described in Figures 8, 9, and 10 is done.

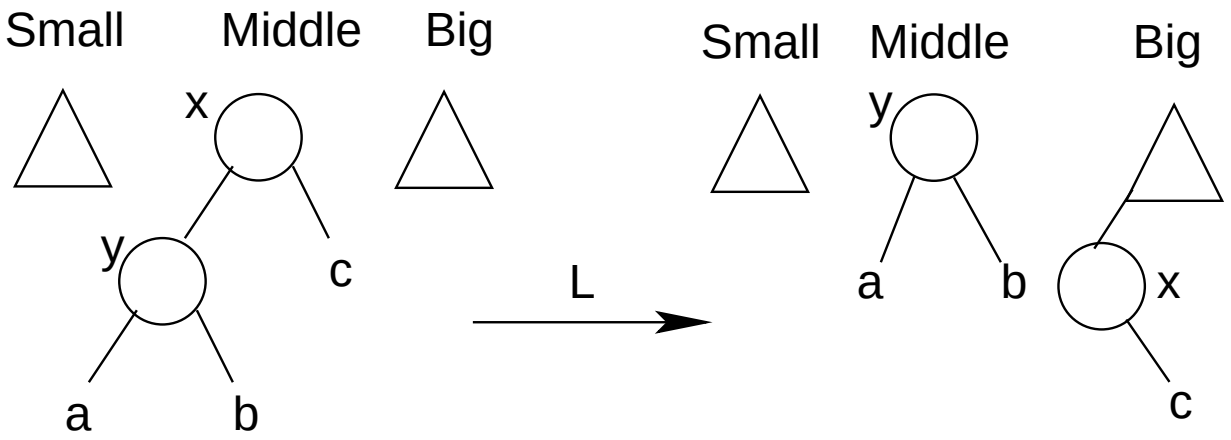


Figure 8: L rotation for top-down splay

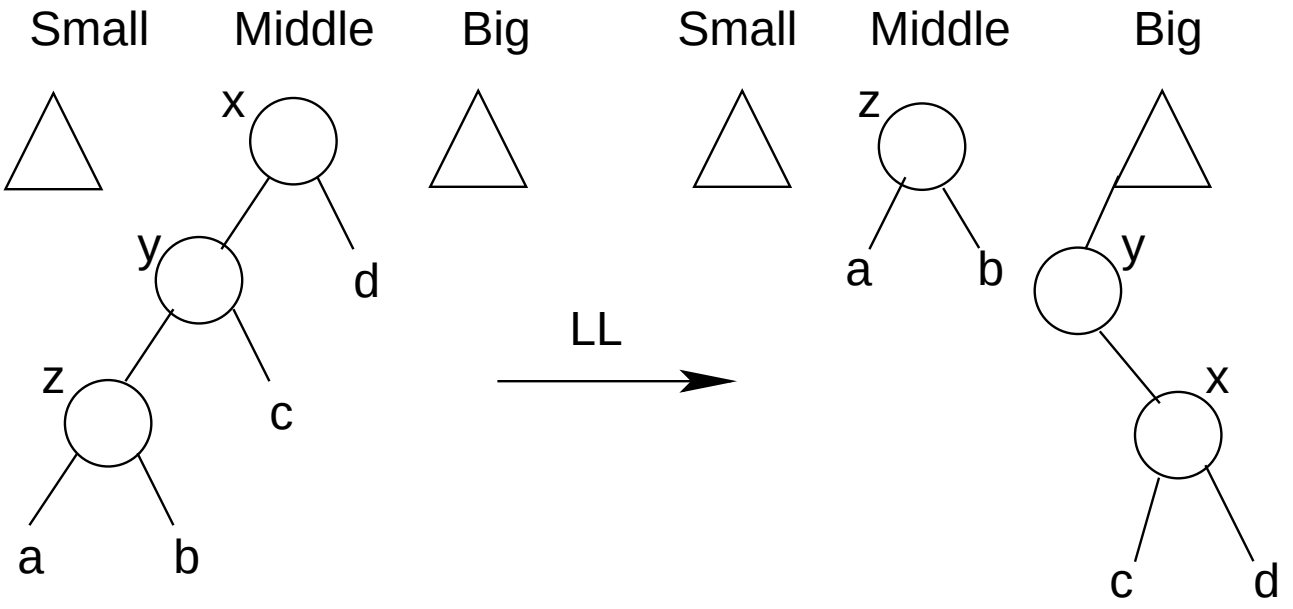


Figure 9: LL rotation for top-down splay

In Figure 8, x is the current node and y is the node we are moving to (this is a one-level, and hence

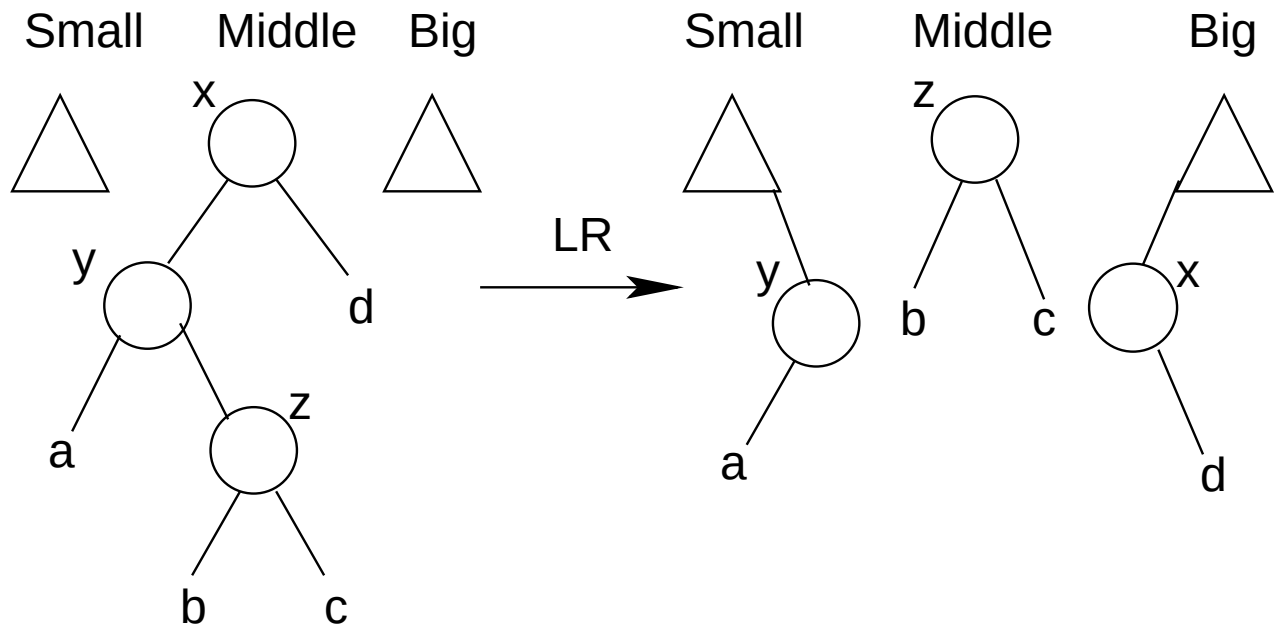


Figure 10: LR rotation for top-down splay

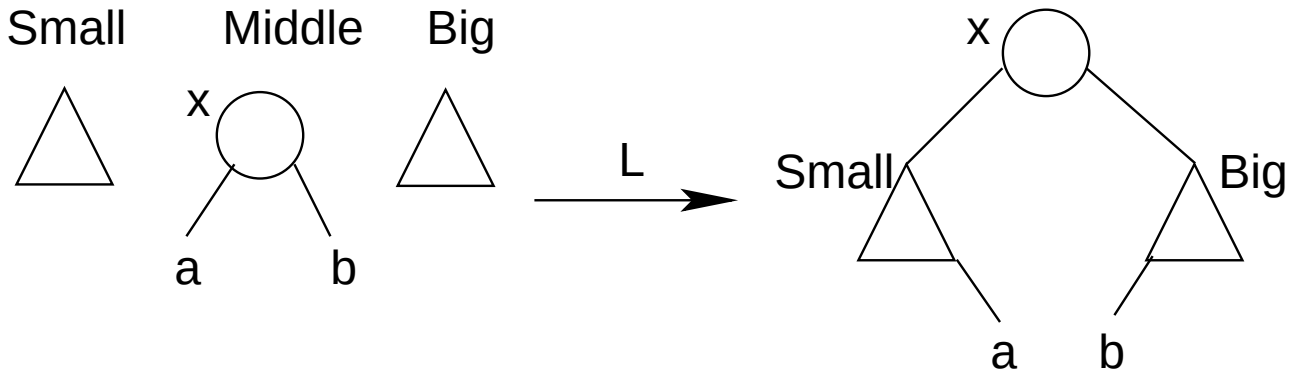


Figure 11: Completion step for top-down splay

final, move). The L rotation is done only when y will be a full node after the operation. When y will be a nonfull node after the operation, we proceed directly to the completion step shown in Figure 11. As can be seen, the L splay step does not transform any leaf nodes of $Middle$ into nonleaf nodes of Big.

In Figures 9 and 10, x is the current node and z is the node two-levels down that we are moving to. Once again, the shown splay step is done only when z will be a full node following the operation. When z will not be a full node following the operation, a one-level move to y is made employing an L-type splay step (see Figure 8). This one-level move is followed by the completion step of Figure 11.

5.3 Properties

Since one element per node splay trees provide $O(\log n)$ time per operation performance only in the amortized sense, we are concerned with the amortized complexity of $buSplay(m)$ and $tdSplay(m)$ only. By modeling operations in a $buSplay(m)$ and a $tdSplay(m)$ that terminate at a nonfull leaf node as unsuccessful operations performed on a one element per node splay tree that coresponds to the residual tree of the $Splay(m)$, we can use the one element per node splay tree proofs to conclude that the amortized number of splay steps per operation is $O(\log n)$. Hence, the amortized complexity of the search and insert operations is $O(\log n + m)$, and that of the delete operation is $O(m \log n)$.

The space analysis for $tdSplay(m)$ is the same as that for $BST(m)$. For $buSplay(m)$, we include a `parent` field, because the inclusion of such a field improves run time performance. Now, the size of a supernode is $8m + 16$ bytes, and that of a $buSplay(1)$ node is 20 bytes.

Theorem 5.1 *A $buSplay(m)$ with n-pairs requires at least $(n/m)(8m + 16)$ bytes, and at most approximately $16(m + n)$ bytes, when $m >> 1$.*

Proof Same as that of Theorem 2.2. ■

The worst-case *space ratio* for an n-pair $buSplay(m)$ is

$$\begin{aligned} SR(m,n) &\approx \frac{20n}{16(m+n)} \\ &= \frac{5n}{4(m+n)} \end{aligned} \tag{6}$$

From Equation 6, we see that when $m < 0.25n$, an n-pair $buSplay(m)$ takes less space than does an n-pair $buSplay(1)$. For $n >> m >> 1$, $buSplay(m)$s provide a worst-case space ratio of 1.25. Comparing best cases for $buSplay(m)$s and $buSplay(1)$s gives a space ratio of $20n/((n/m)(8m+16)) = 5m/(2m+4) \approx 2.5$.

6 Reducing Cache Misses

The performance of the dynamic search tree operations: insert, delete and search, can be improved by enhancing cache utilization. Let's consider doing this for our supernode structures. The frequently used

fields of the supernode are `LeftChild`, `RightChild`, `minKey`, and `maxKey`; these four fields are accessed by all three operations during the root to leaf traversal phase (the `parent` field, by contrast, is used only by the insert and delete operations during the bottom up rebalancing pass, and by bottom-up splay trees during the bottom to top splay pass). So, we use the node layout shown in Figure 12.

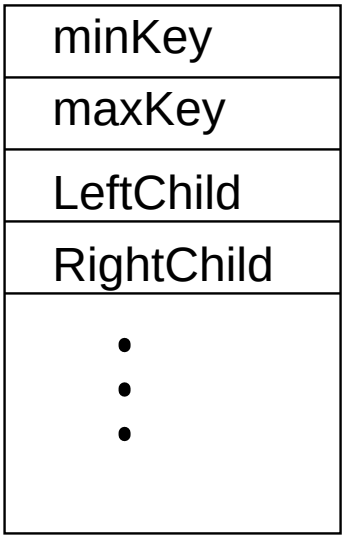


Figure 12: Supernode layout for the efficient use of cache memory

With this layout we can access all four of the frequently accessed fields with at most one cache miss provided all four fields fit into the same cache line. This happens, for example, when the supernode begins at a cache line boundary and the memory required by the above four fields is no more than the size of a cache line.

Typically, allocating memory that is aligned to a cache line boundary is more time consuming than allocation of unaligned memory. Therefore, when using cache aligned memory, we allocate aligned memory in chunks that are large enough for several supernodes.

Figure 13 shows a cache aligned memory chunk that is the size of 3 supernodes. If the cache line size is $LineSize$, then each supernode is cache aligned iff

$$NodeSize \bmod LineSize = 0 \tag{7}$$

where $NodeSize$ is the number of bytes allocated to a supernode. Since $LineSize$ is typically a power of 2 (the size of an L1 cache line on a SUN Sparc workstation is 32 bytes and that of an L2 cache line is 64 bytes), for Equation 7 to hold, $NodeSize$ must be a power of 2 and $\geq LineSize$. From our earlier space analyses, we know that for $AVL(m)$, $RB(m)$, and $buSplay(m)$

$$NodeSize = 8m + 16 \tag{8}$$

From Equations 7 and 8, we get

$$(8m + 16) \bmod LineSize = 0 \tag{9}$$

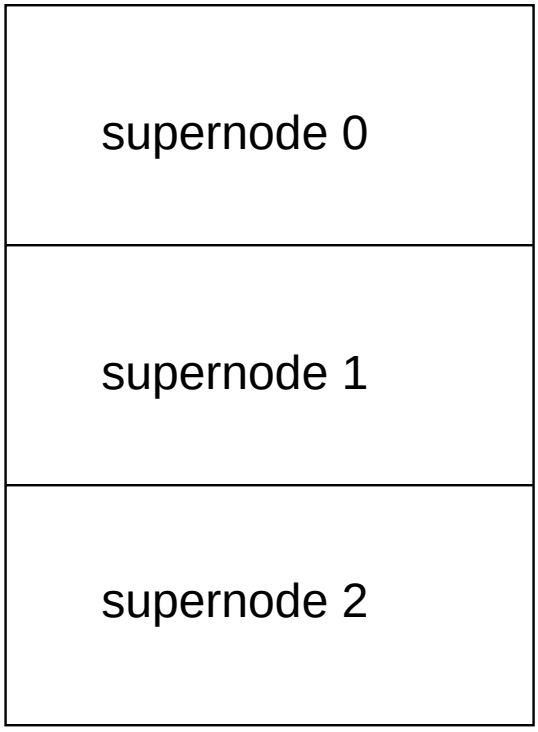


Figure 13: Cache aligned chunk with 3 supernodes

Cache alignment of all supernodes in a cache aligned chunk is assured, for example, when $m = 2$, 6, 10, 14, ... and $LineSize = 32$ bytes.

In the case of $tdSplay(m)$, however, the node size is $8m + 12$ bytes, which isn't a multiple of 32 for any value of m. To assure cache line alignment of each supernode, we must, for each m, waste some space and make the supernode size a multiple of 32. In our implementation of cache aligned $tdSplay(m)$ trees, we add a 4-byte field to each supernode. So, the node size becomes $8m + 16$ bytes. Hence, the modifed node size is a multiple of 32 for $m = 2$, 6, 10, ...

Next, suppose we have a direct-mapped cache as is the case for a SUN Sparc workstation. Byte b of memory maps into cache line

$$CacheLine(b) = \lfloor b/LineSize \rfloor \bmod NumberOfCacheLines \tag{10}$$

Suppose we use cache aligned supernodes and that each supernode is an integral number of cache lines as suggested in Equation 7. Suppose that our first supernode begins at byte 0. Then, assuming no memory is allocated for other purposes, each of our supernodes begins at byte $i * NodeSize$ for some non-negative integer i. Therefore, the frequently used 4 fields of each supernode fall into cache lines given by

$$\begin{aligned} &\lfloor i * NodeSize/LineSize \rfloor \bmod NumberOfCacheLines \\ &= (i * LinesPerNode) \bmod NumberOfCacheLines \end{aligned} \tag{11}$$

where $LinesPerNode = NodeSize/LineSize$ is the number of cache lines needed for a supernode.

Since the number of cache lines in both the L1 and L2 cache of a computer is a power of 2, we see from Equation 11 that when $LinesPerNode$ is even, the four frequently used fields of a supernode are always mapped into an even cache line. Effectively, only half the cache lines get used.

From the following theorem, it follows that as long as $LinesPerNode$ and $NumberOfCacheLines$ are relatively prime, the numbers generated by Equation 11 are uniformly distributed over $\{0, 1, 2, \ldots, NumberOfCacheLines\}$.

Theorem 6.1 *For integer j ($0 \leq j < n$), $(j * q)$ mod n generates precisely d copies of the set {0, d, 2d, ..., n - d} of the n/d numbers, where $d = gcd(q, n)$. ([1], pp. 129-130)*

From this theorem, it follows that for every q and every n such that q ($1 \leq q < n$) and n are relatively prime (that is, $d = 1$), the set {0, 1, 2, ..., $n - 1$} is generated by $(j * q)$ mod n, where $0 \leq j < n$.

Therefore, for $0 \leq j < \infty$,

$$(j * q) \bmod n \tag{12}$$

where q and n are relatively prime, generates uniformly distributed numbers over 0, 1, 2, ..., $n - 1$.

Since $NumberOfCacheLines$ is a power of 2, it follows that by choosing $LinesPerNode$ to be an odd number we ensure Equation 11 gives a uniform distribution of cache lines.

Note that choosing $LinesPerNode$ such that $k * LinesPerNode$ and $NumberOfCacheLines$ are relatively prime, where k is an integral constant > 1 also gives a uniform utilization of cache lines. So, for example, when $LinesPerNode = 6.5$, we get a uniform utilization of cache lines.

When a cache line is 32 bytes, cache utilization may be further improved by separating the frequently used fields of a node from the non-frequently used fields. When this is done, we can arrange for the frequently used fields of two nodes to fill a cache line. Doing this complicates the code somewhat, and we do not pursue this option further in this paper.

7 Experimental Results

To obtain an experimental evaluation of the supernode schemes relative to their single pair per node counterparts, we programmed the various dictionary structures described in this paper in C++. The codes were compiled using the Gnu C++ compiler `g++` with maximum optimization mode `O3` and run on a SUN Ultra Sparc Station in which each L1 cache line is 32 bytes and the total amount of L1 cache is 16K bytes. The amount of main memory on this station was 2GB.

7.1 Memory Requirements

To determine the performance of the supernode schemes relative to their one pair per node counterparts, we inserted n randomly generated pairs into an initially empty dictionary and measured the total memory used by the resulting dictionary. Table 1 gives the results for the cases n = 1M (1 million) and n = 4M. For each n, 10 different and randomly generated sequences of n pairs were used. The average of

the memory requirements for these 10 sequences is shown in Table 1. For each of our 20 test cases, the supernode version of a dictionary structure took considerably less space than did the one pair per node version. The measured average space ratio for $AVL(26)$, for example, is $24/12.1 \approx 2$ when $n = 1$M and is $96/48.7 \approx 2$ when $n = 4$M. The ratio of 2 is within the range [1.5, 3] predicted by our analysis of Section 3.3. The observed space ratios for $RB(m)$ and $buSplay(m)$ are almost the same as those for $AVL(m)$. For $tdSplay(m)$, the space ratios are smaller. This is not surprising given that our analysis of Section 5 predicts that this ratio will be between approximately 1 and 2 (rather than between 1.5 and 3). The measured space ratio for $tdSplay(26)$ is about 1.3.

Table 1: Actual memory utilization in mega bytes

data set	m	$AVL(m)$	$RB(m)$	$buSplay(m)$	$tdSplay(m)$
1M	01	24.0	24.0	24.0	16.0
	06	14.2	14.2	14.3	14.3
	12	12.9	12.9	13.0	13.0
	22	12.3	12.3	12.3	12.3
	26	12.1	12.2	12.2	12.2
4M	01	96.0	96.0	96.0	64.0
	06	56.6	56.6	56.9	57.0
	12	51.6	51.6	51.8	51.8
	22	49.2	49.2	49.3	49.3
	26	48.7	48.7	48.8	48.8

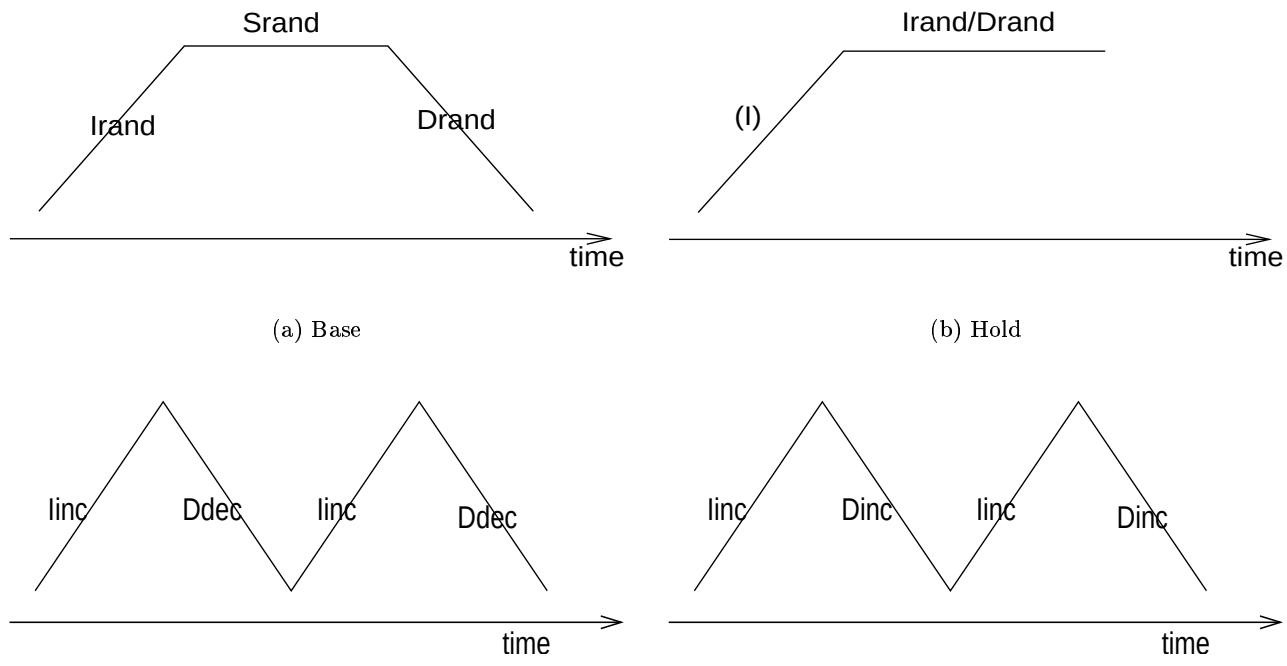
7.2 Run Times

7.2.1 Models

For the run time tests, we used the four test models (base, hold, stack, and queue) that are as shown in Figure 14, and a histograming application [5].

1. **[Base Model]** In the base test model (Figure 14(a)), n pairs with randomly generated but distinct keys in the range [1,2n] are inserted into an initially empty dictionary and the total time taken to make these n insertions is measured. Next, a random sequence of n searches for pairs with keys in the range [1,2n] is done, and the total time for these n searches is measured. Note that some of these n searches will be unsuccessful (i.e., searches for keys that are not in the dictionary). Finally, the n pairs in the dictionary are removed in some random order, and the time for all n deletions is measured.

2. **[Hold Model]** In the hold model [3], as can be seen in Figure 14(b), we start with a dictionary that has n pairs and perform an intermixed sequence of insert and delete operations. This sequence



(a) Base

(b) Hold

(c) Stack

(d) Queue

Figure 14: Four test models (I: Insert, S: Search, D: Delete, rand: random sequence, inc: increasing sequence, dec: decreasing sequence)

of operations is generated randomly so that the size of the dictionary remains at roughly n. The time to perform the intermixed sequence of q insert and delete operations is measured.

3. **[Stack Model]** In the stack model (Figure 14(c)), we start with an initially empty dictionary and insert n pairs in increasing order of key. Then these pairs are removed in decreasing order of key. The insert/delete cycle is repeated once. The time taken for the $4n$ operations is measured.

4. **[Queue Model]** The queue model (Figure 14(d)) differs from the stack model only in that the deletes are done in the same order as the inserts (i.e., in increasing order of key).

For each model and each choice of n and m, run times were measured using 10 different data sets. The average of these times is reported in Section 7.2.2.

The run time performance of the single element per node version of a data structure was compared with that of the (non cache aligned) supernode as well as the cache aligned supernode (each supernode begins at a cache line boundary) version of the same data structure. Notice that the single element per node versions of our data structures do not use nodes that necessarily start at cache line boundaries.

The non-cache aligned schemes used the new and delete methods of C++ to get and free nodes. For the cache aligned schemes, nodes were allocated from large chunks of cache aligned memory (this was done to reduce memory allocation time). The chunk size was chosen so as not to exceed the size of L1 cache and so as to be a multiple of the node size.

$$\begin{aligned} ChunkSize &= Nnodes * LinesPerNode \\ &= \lfloor \frac{NumberOfCacheLines}{LinesPerNode} \rfloor * LinesPerNode \end{aligned} \tag{13}$$

where $Nnodes$ is the number of nodes that can be allocated from a memory chunk.

7.2.2 Results

The cache aligned versions of our supernode schemes outperformed the non cache-aligned schemes when $m < 50$. For larger values of m, the added cost of cache aligned memory allocation, at times, resulted in poorer performance than exhibited by the non cache-aligned schemes. Using cache-aligned nodes generally reduced the run time by about 20% for small m (say $m < 10$). This reduction diminished with increasing m, and even became negative, in some tests, when $m > 50$.

Because of space limitations, we report the run time results only for our cache-aligned implementations with $m = 2$, 6, 10, 18, 26, 34, 42, 50, 58, and 66. Each supernode is allocated from a cache-aligned chunk whose size is given by Equation 13. From the development of Section 6, we know that all supernodes allocated in this fashion are cache aligned for the stated m values. Furthermore, from Theorem 6.1, it follows that when $m = 2$, 10, 18, 26, 34, 42, 50, 58, or 66, we get a uniform distribution of cache lines. For comparison purposes, the run times for conventional the $m = 1$ implementation using both non-aligned and aligned memory also are presented.

Base Model Run Times

For the base model, we experimented with n =10,000 (10K), 100K, 1M (million), and 4M. The observed run times for insert, search, and delete portions of the experiment are shown in Figures 15 through 26. These figures use the following abbreviations: RBB (bottom up red-black trees), SPB (botton up splay trees, and SPT (top down splay trees).

For the case $m = 1$, there are two points plotted for each data structure (3 points for red-black trees). The point with the larger run time corresponds to the case when no attempt is made to ensure that nodes fall on cache line boundaries. The point with the smaller run time corresponds to the case when memory for each node is allocated so that each node begins at a cache line boundary (of course, this results in some wasteage of memory). In the case of red-black trees, the third point associated with $m = 1$ gives the run time for the STL implementation (`http://www.sgi.com/Technology/STL/`) of red-black trees. The time to insert elements using the STL implementation of red-black trees was consistently between the times taken by our non cache aligned and cache aligned implementations. For searches and deletes, the STL implementation was always slower than our non aligned and aligned implementations. Note that although the STL implementation does not use cache aligned nodes, it allocates nodes by using the C++ `new` method to get a chunk of memory large enough for (say) 40 nodes and suballocates individual node memory from this chunk. Memory chunks are not deallocated using C++'s `delete` method. Instead, the STL implementation keeps track of free memory within a chunk and reallocates this as needed. Using this strategy, the number of invocations of `new` is dramatically reduced and the number of invocations of `delete` is zero. This leads to a reduction in overall memory allocation time, which in turn benefits the insert operation. The impact on overall deallocation time is minimal because memory deallocation is relatively fast using either scheme.

The following conclusions may be drawn from this set of experiments:

1. AVL and red-black trees are very competitive in their performance. Both clearly outperform both varieties of splay trees used by us. Top down splay trees provide superior performance relative to bottom up splay trees.

2. The use of supernodes not only saves memory (as shown in Table 1), but also leads to reduced run time (provided the supernode size m is chosen properly).

3. Although the run time reduction in going from non cache aligned nodes with $m = 1$ to cache aligned nodes with $m = 1$ is more significant than the further reduction obtained in going from $m = 1$ cache aligned nodes to cache aligned supernodes ($m > 1$), the former reduction in run time comes at the expense of increased memory utilization, whereas the latter run time reduction comes with reduced memory utilization.

4. For AVL and red black trees, the best run time is generally obtained when $m = 6$. However, the run time when $m = 6$ isn't much less than that when $m = 26$. The latter choice of m results in reduced memory requirements and may represent a good compromise for applications in which

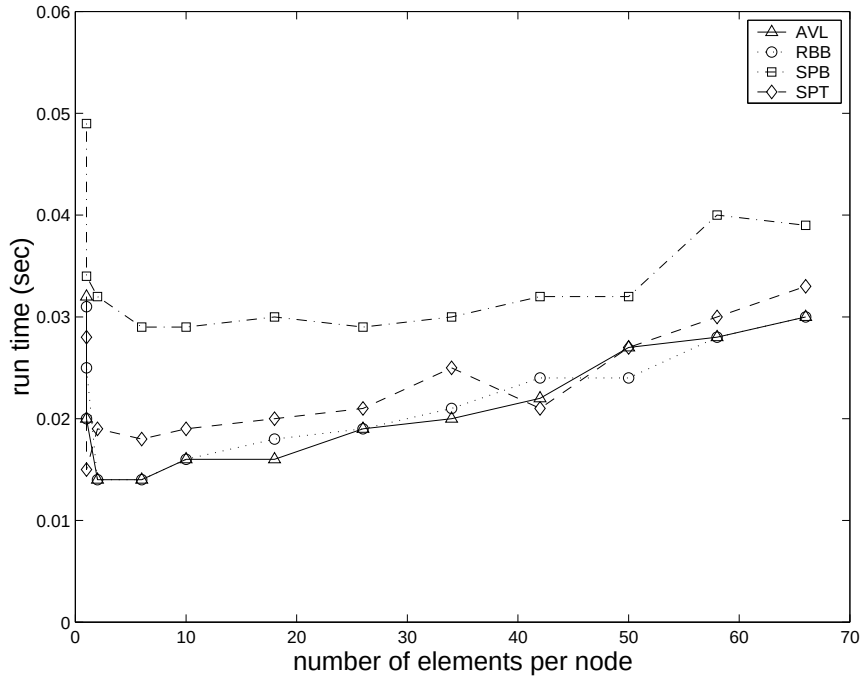


Figure 15: Insert time in base model: $n = 10\text{K}$

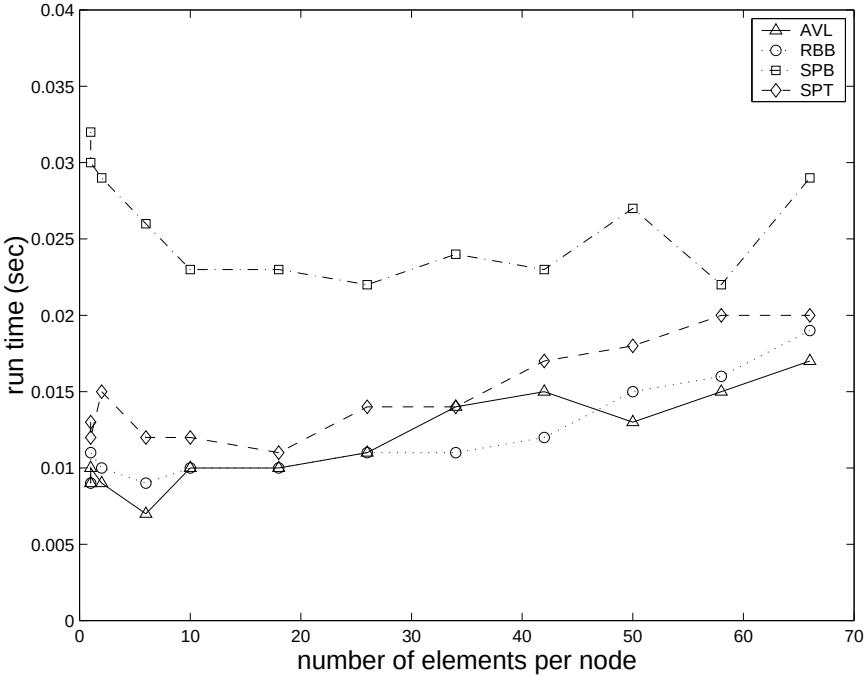


Figure 16: Search time in base model: $n = 10\text{K}$

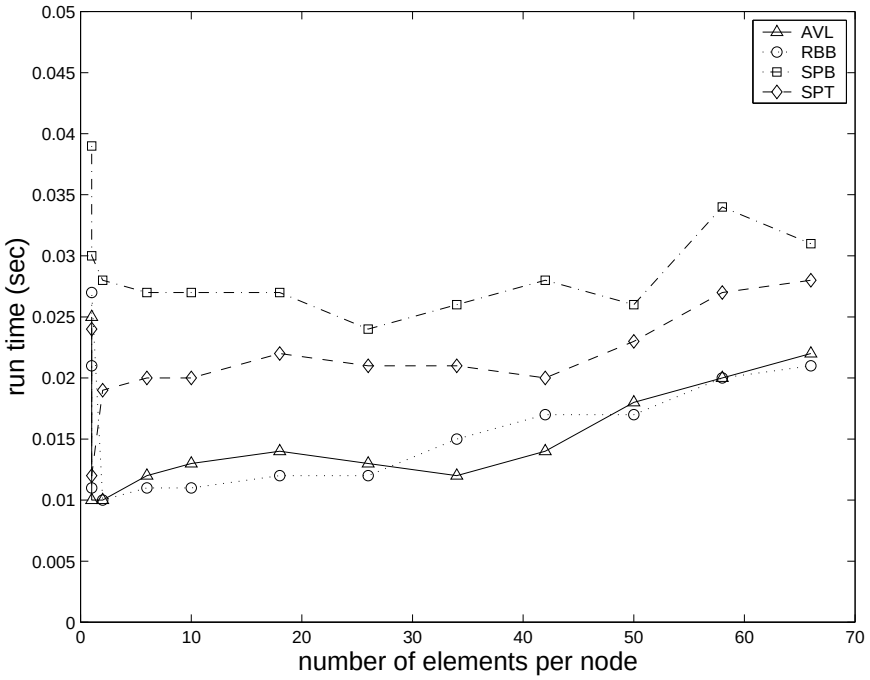


Figure 17: Delete time in base model: $n = 10$K

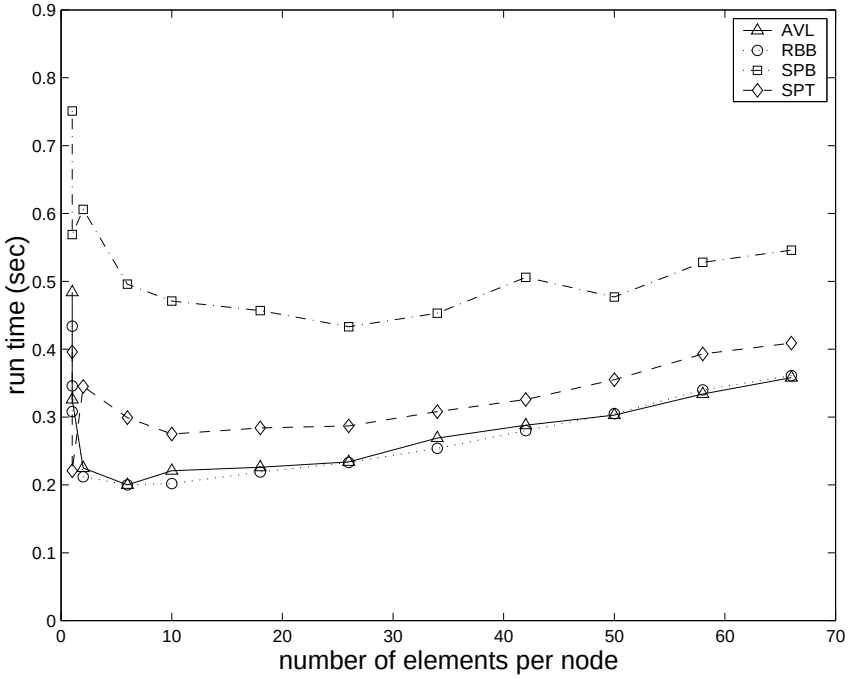


Figure 18: Insert time in base model: 100K data

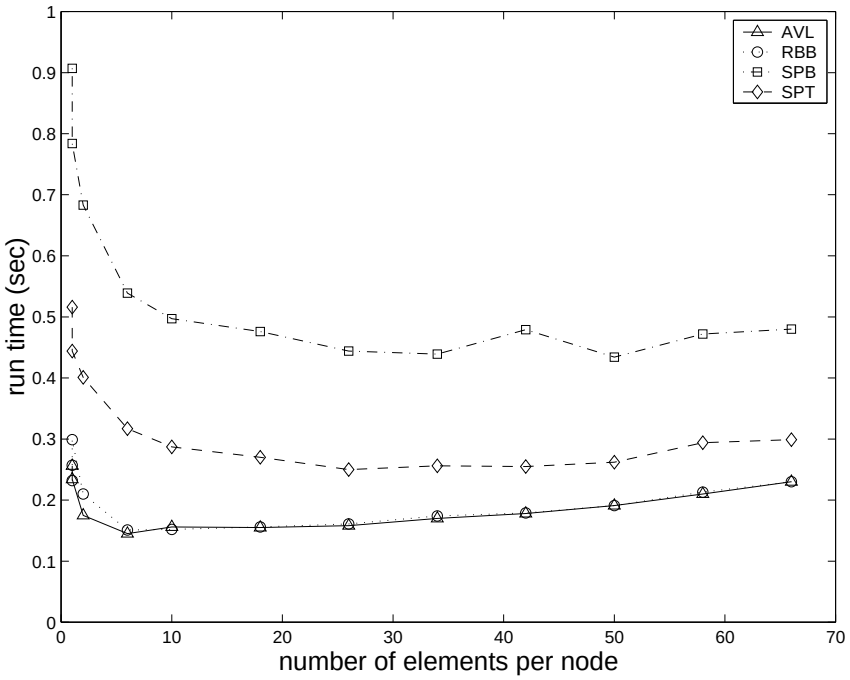


Figure 19: Search time in base model: $n = 100\text{K}$

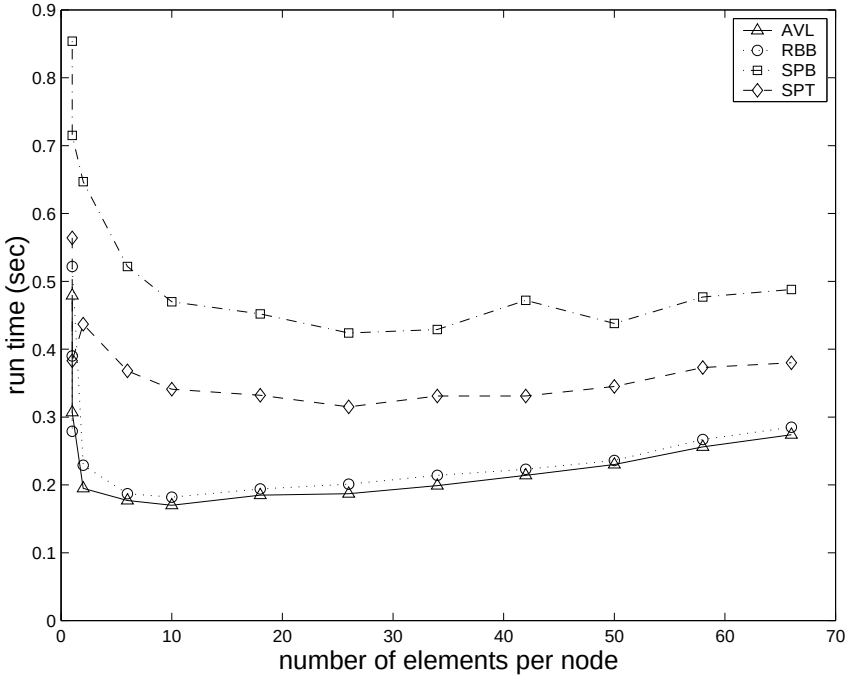


Figure 20: Delete time in base model: $n = 100\text{K}$

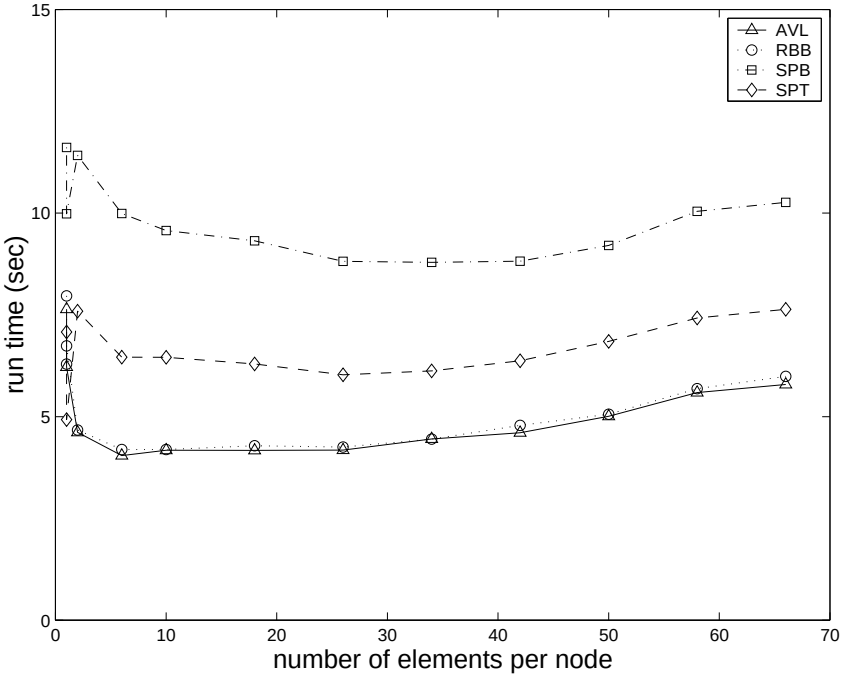


Figure 21: Insert time in base model: $n = 1\text{M}$

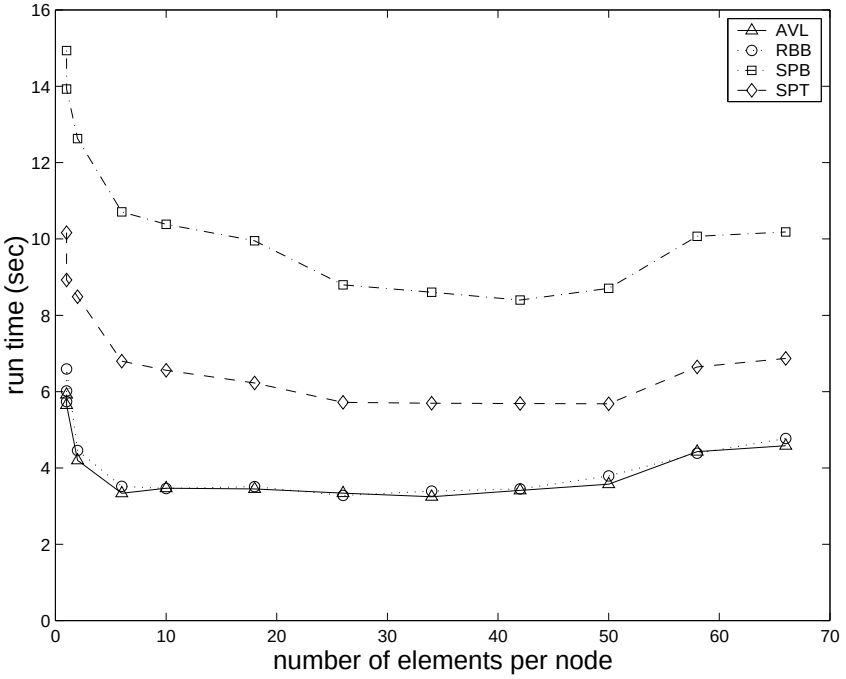


Figure 22: Search time in base model: $n = 1\text{M}$

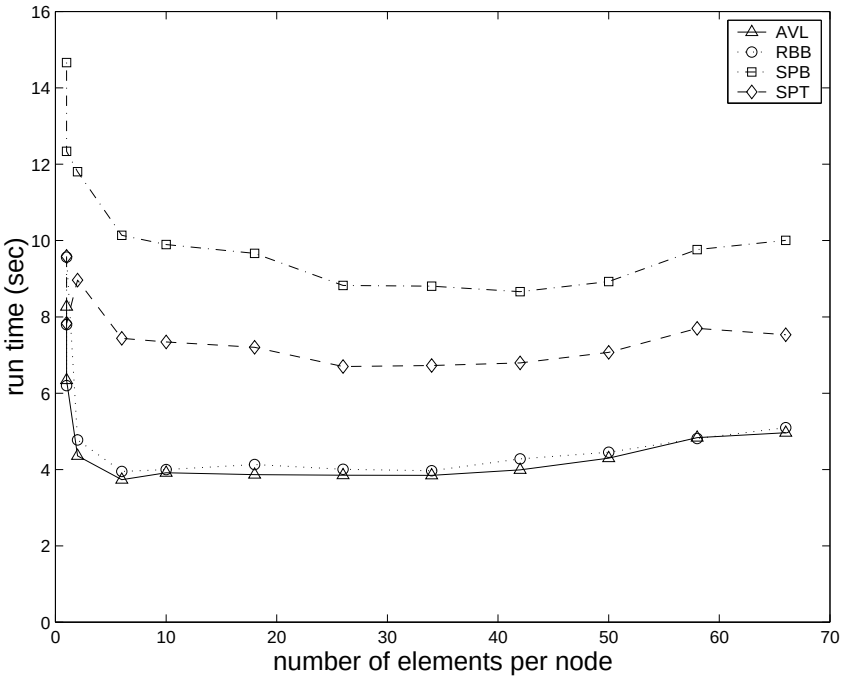


Figure 23: Delete time in base model: $n = 1\text{M}$

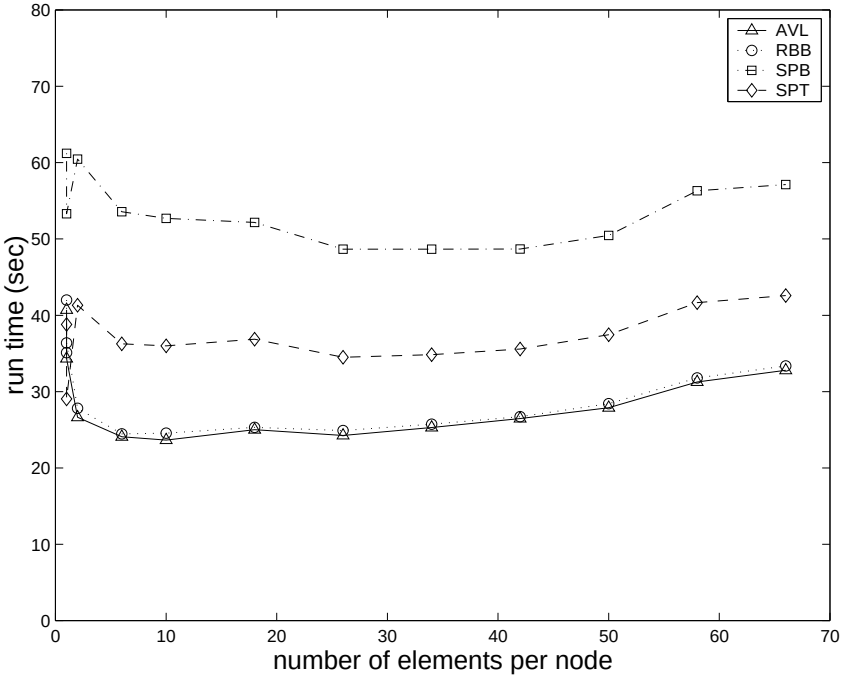


Figure 24: Insert time in base model: $n = 4\text{M}$

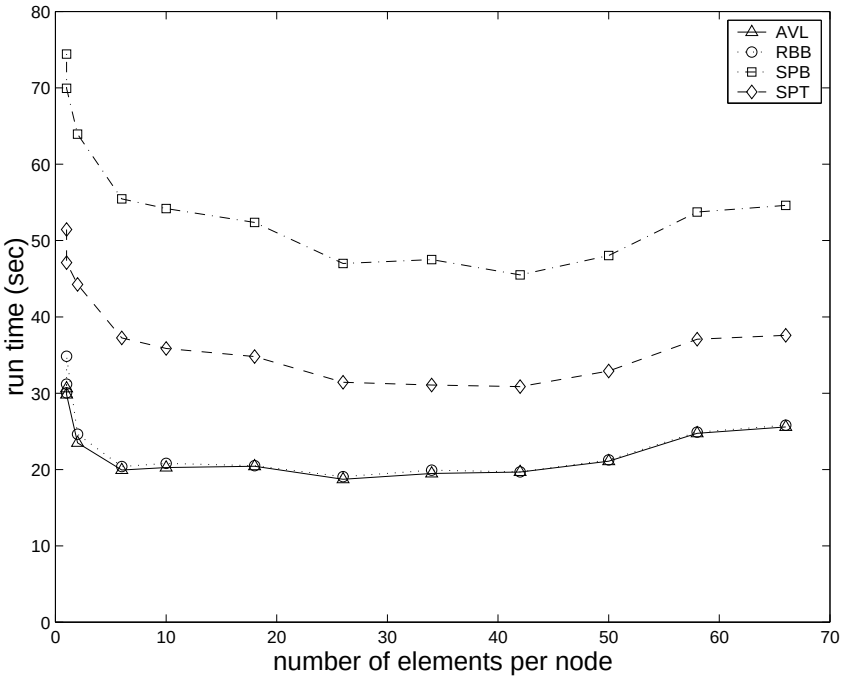


Figure 25: Search time in base model: 4M

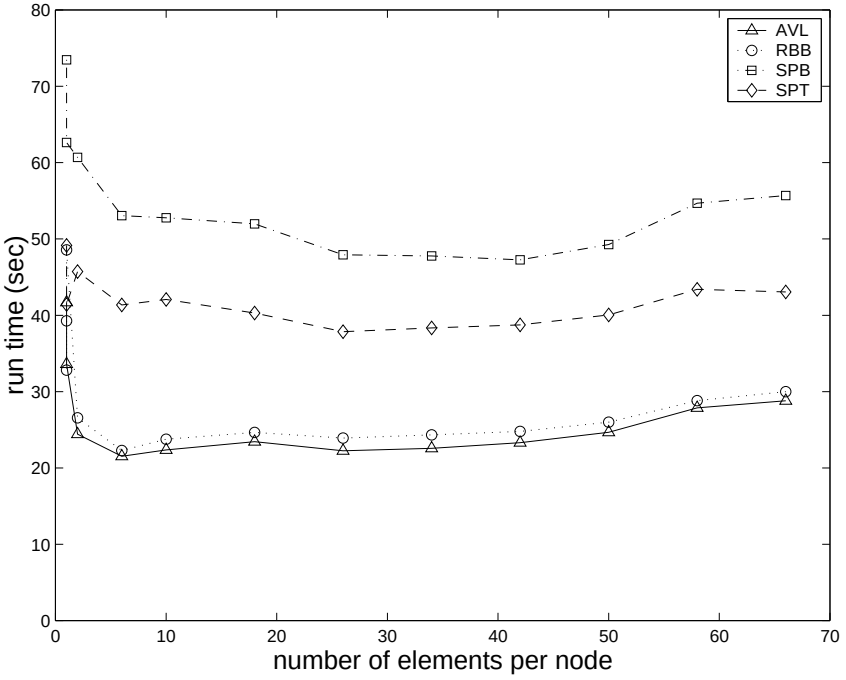
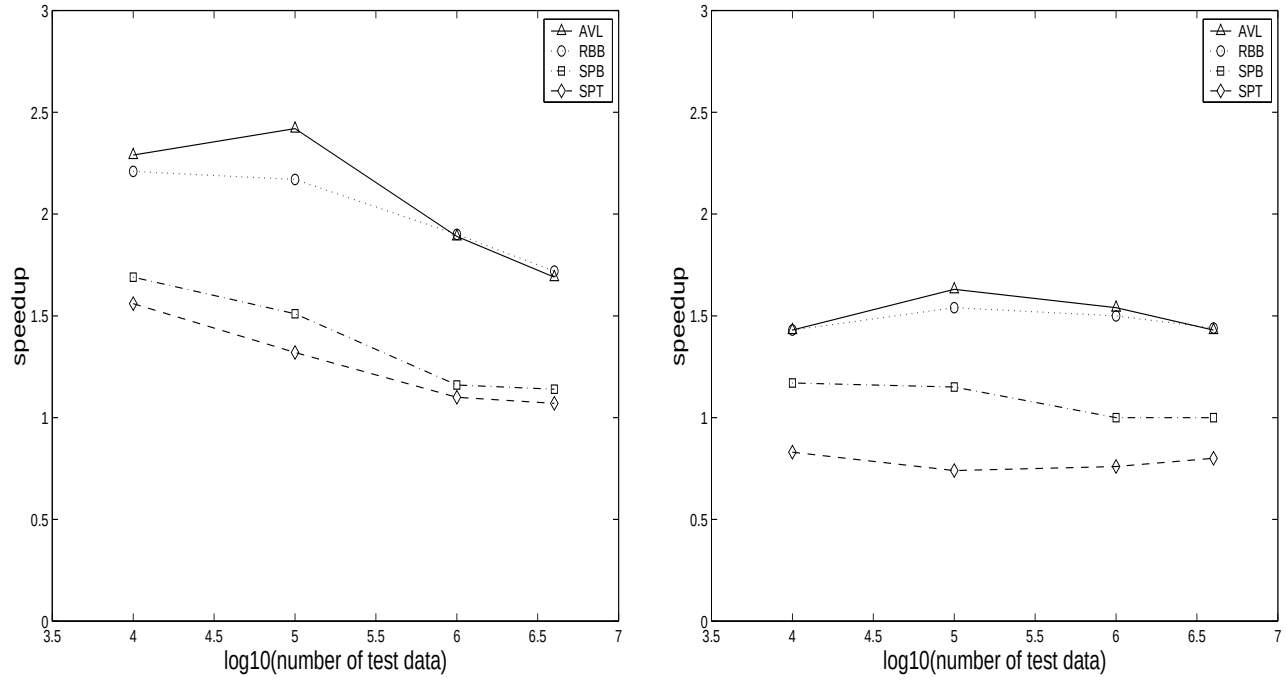


Figure 26: Delete time in base model: $n = 4$M

both memory and time need to be reduced. For both varieties of splay trees, $m = 26$ generaly gave best run time performance.

5. Figures 27 through 29 show the speedup obtained by our supernode scheme when $m = 6$. The reported speedup is shown relative to cases when $m = 1$ and non aligned memory is used as well as when $m = 1$ and cache aligned memory is used. Speedup is defined to be the ratio of the run time of the $m = 1$ scheme divided by the run time for the supernode scheme. The x-axis is drawn to a logarithmic scale.



(a) relative to m=1 and non aligned

(b) relative to m=1 and aligned

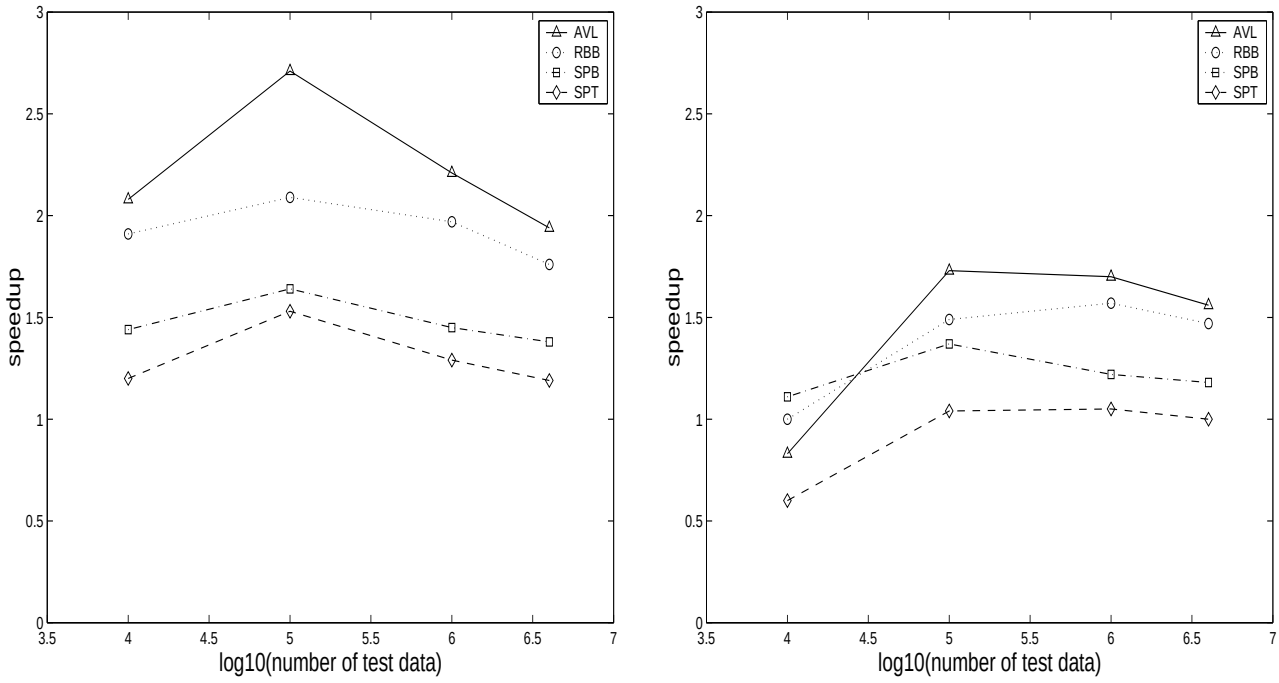
Figure 27: Speedup of insert operations in the base model when m = 6

Hold Model Run Times

Recall that for the hold model, n is the initial size of the dictionary and that an intermixed sequence of q insert/delete operations is performed. The operation sequence is designed so that the size of the dictionary remains roughly at n as the operations are performed. The (n, q) pairs that we experimented with are (100, 1M), (1K, 1M), (10K, 1M), (100K, 1M), (1M, 1M), and (4M, 4M). The run times for these pairs are shown in Figures 30 through 35.

For the hold model and large n, the observations are quite similar to those made for the base model:

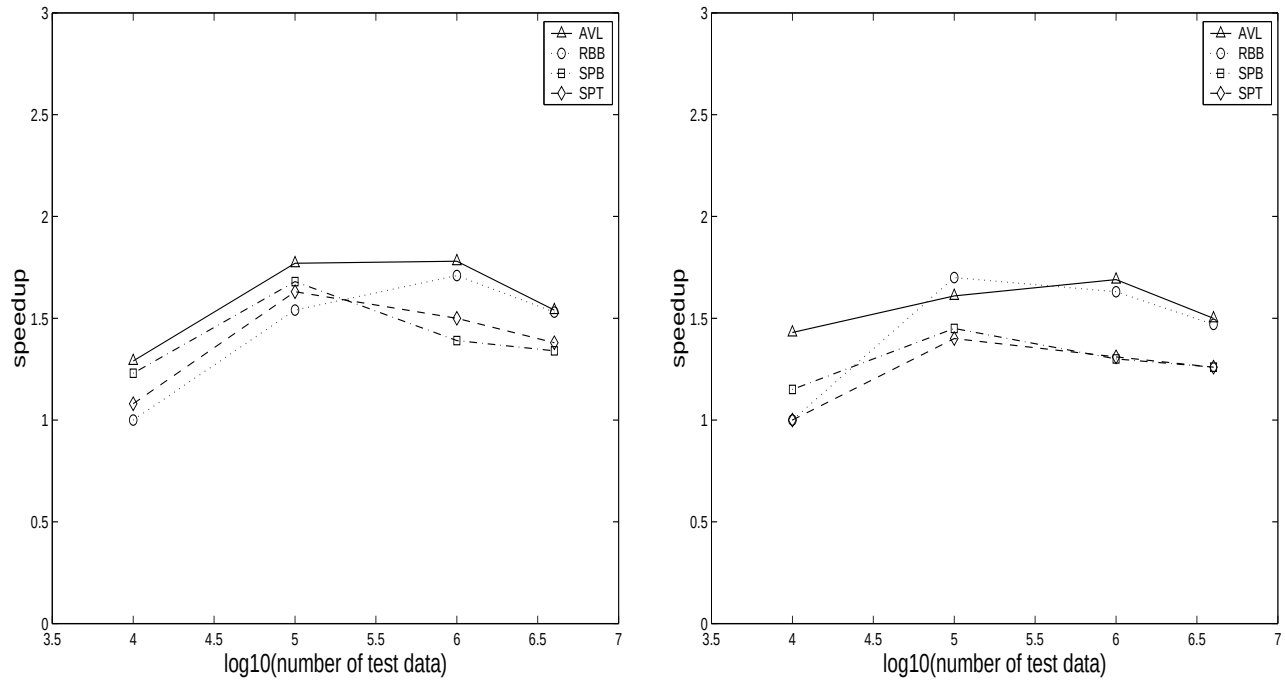
1. For small n (100, 1K, and 10K), top down splay trees generally outperformed the remaining dictionary structures considered by us. There is an apparent anomaly in the run time for bottom up



(a) relative to m=1 and non aligned

(b) relative to m=1 and aligned

Figure 28: Speedup of delete operations in the base model when m = 6



(a) relative to m=1 and non aligned

(b) relative to m=1 and aligned

Figure 29: Speedup of search operations in the base model when m = 6

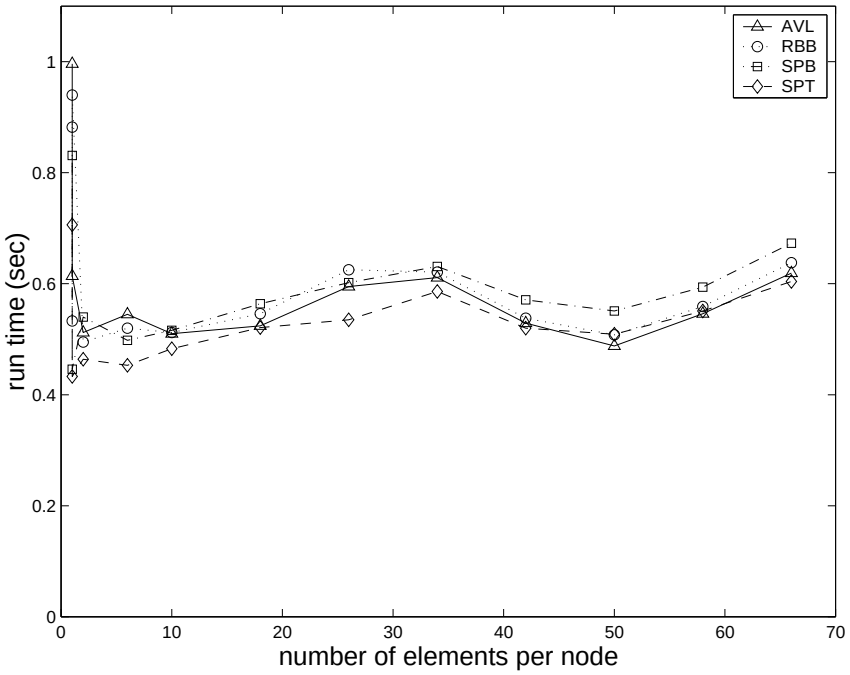


Figure 30: Run time in the hold model: $n = 100$ and $q = 1$M

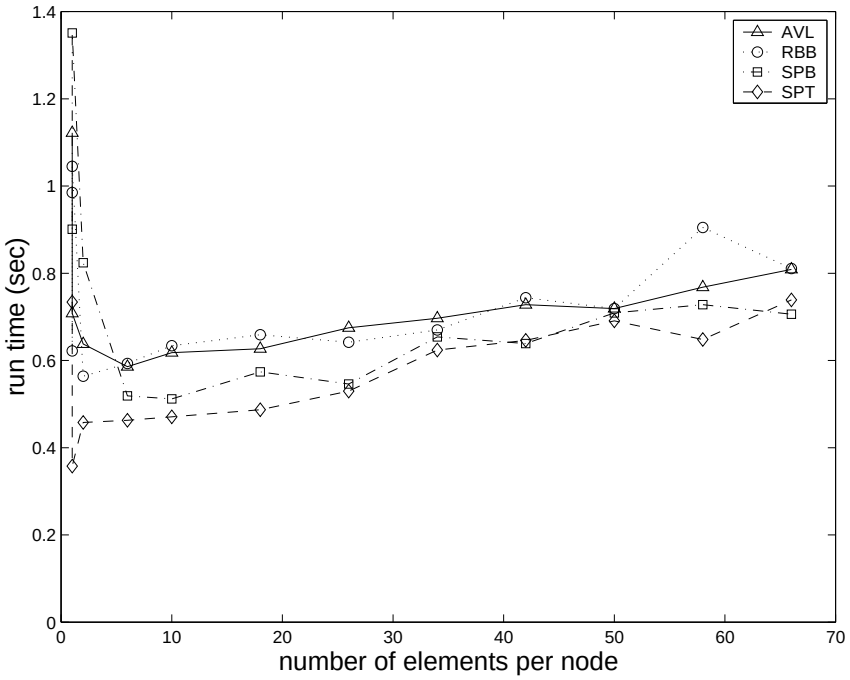


Figure 31: Run time in the hold model: $n = 1\text{K}$ and $q = 1\text{M}$

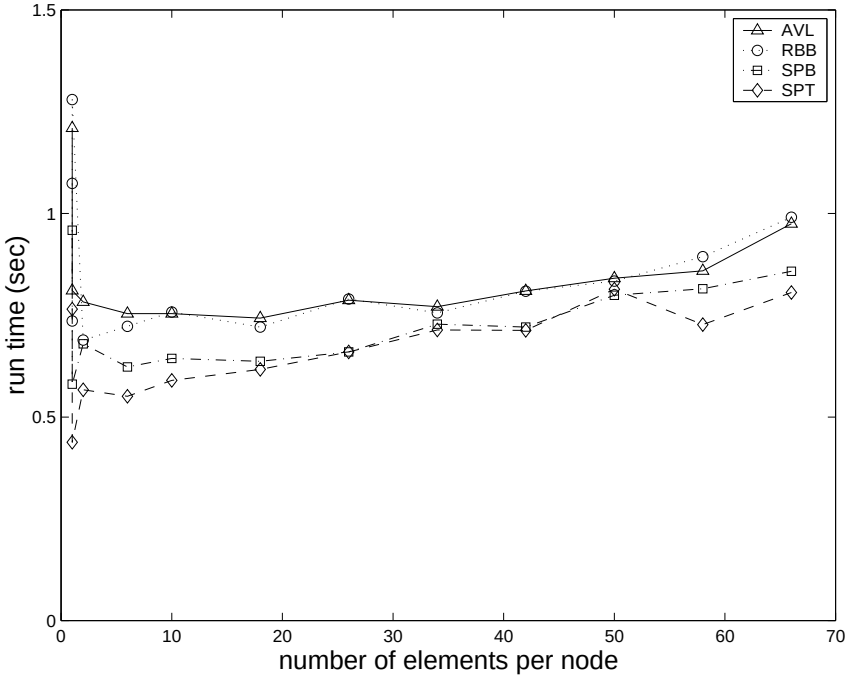


Figure 32: Run time in the hold model: $n = 10\text{K}$ and 1M

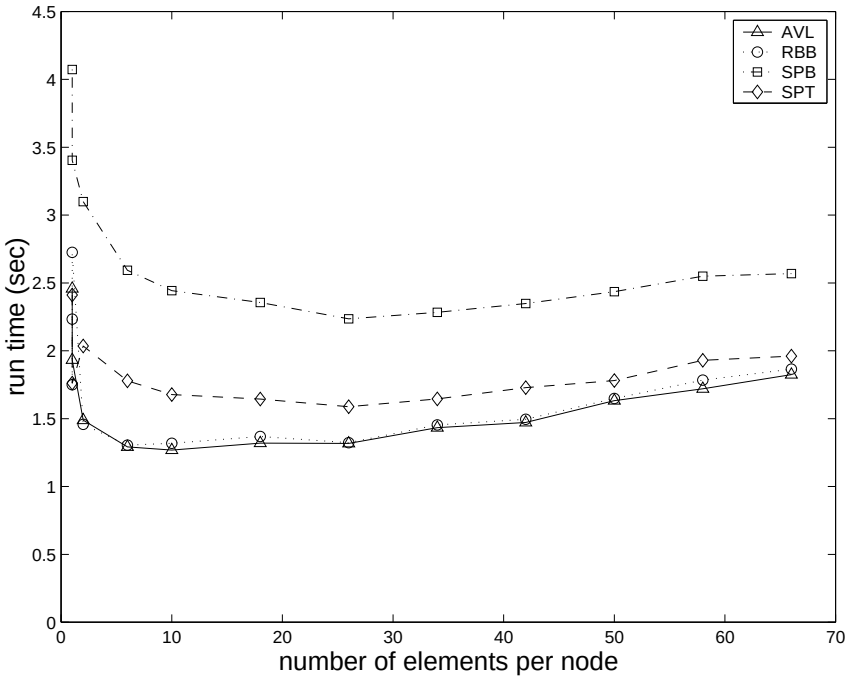


Figure 33: Run time in the hold model: $n = 100\text{K}$ and $q = 1\text{M}$

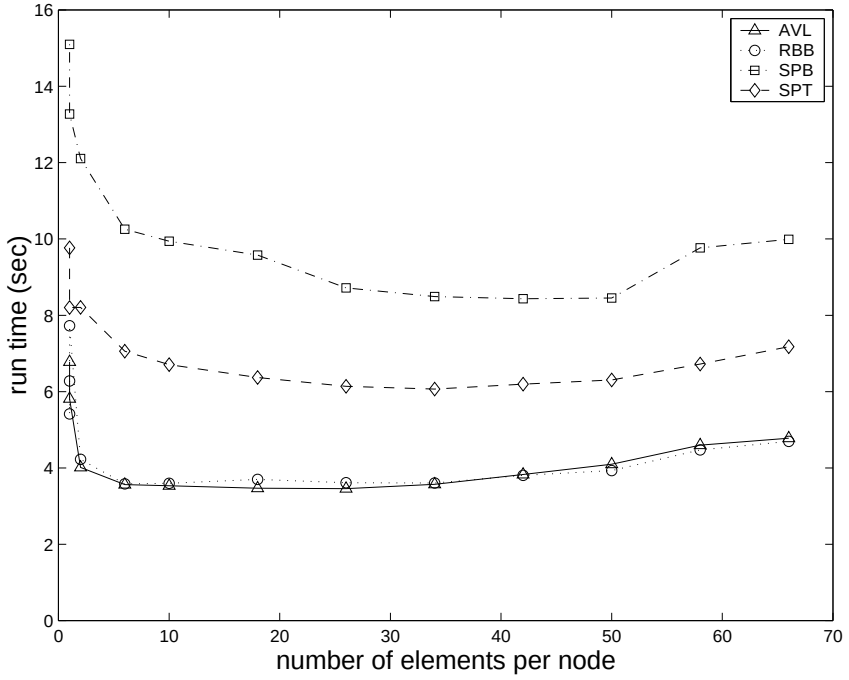


Figure 34: Run time in the hold model: $n = 1\text{M}$ and $q = 1\text{M}$

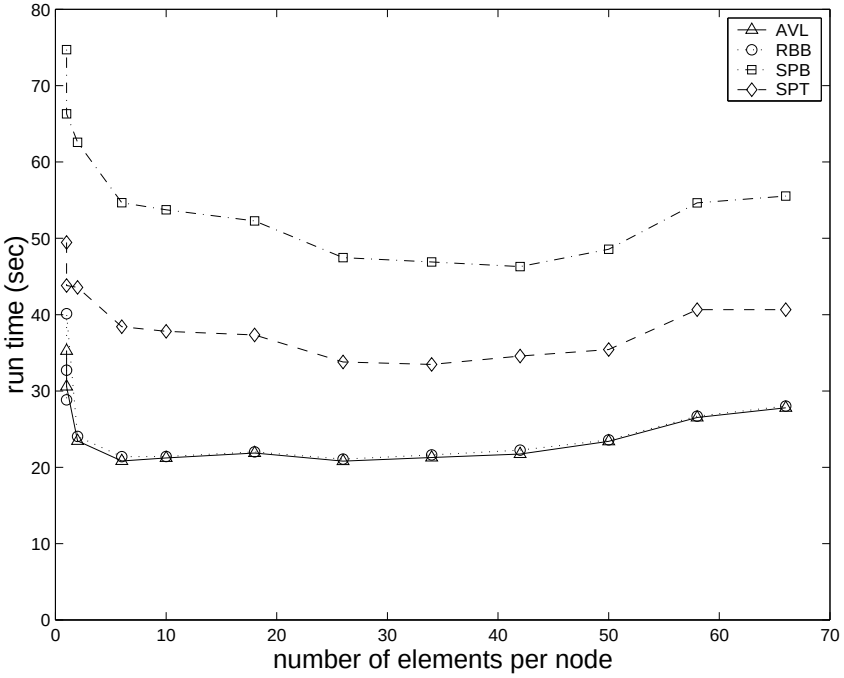


Figure 35: Run time in the hold model: $n = 4$M and $q = 4$M

splay trees when $m = 1$ and $n = 1$K. For this combination of m and n, the run time for the non cache aligned case is smaller than when cache aligned nodes are used. A possible expalantion for this behavior is that the non cache aligned implementation uses 1K nodes that are 20 bytes each, whereas the cache aligned implementation uses 1K nodes that are 32 bytes each. In the former implementation, 80% of the tree fits into the L1 cache, whereas in the cache aligned implementation, only 50% of the tree fits. For smaller n, all of the tree fits in L1 cache, and for large n a very small fraction of the tree fits regardless of whether 20-byte or 32-byte nodes are used.

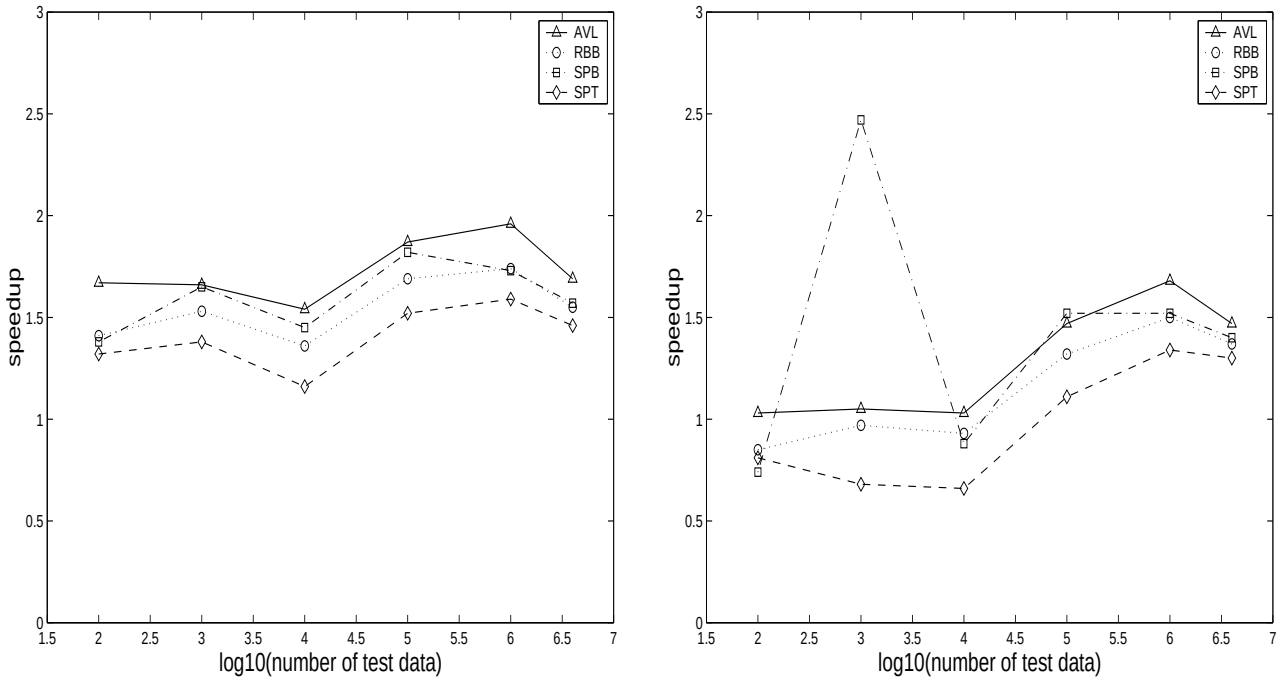
For large n, the relative run time performance is the same as observed for the base model–AVL and red-black trees are very competitive, both clearly outperform top down and bottom up splay trees, and top down splay trees provide superior performance relative to bottom up splay trees.

2. For large n ($n \geq 100$K), $m = 26$ once again represents a good balance between memory saving and run time reduction. Figure 36 shows the speedup obtained by our supernode scheme when $m = 26$.

Queue and Stack Model Run Times

For the queue and stack models, we used $n = 1$K, 10K, 100K, 1M and 4M. The observed run times are shown in Figures 37 through 46. Our observations are:

1. Top down splay trees generally performed the best. They were followed by bottom up splay trees, AVL trees, and red-black trees (in that order).



(a) relative to m=1 and non aligned

(b) relative to m=1 and aligned

Figure 36: Speedup in the hold model when m = 26

2. For top down splay trees, the supernode scheme yields no run time benefits (although one still might use the supernode scheme to reduce memory requirements). The same is true for bottom up splay trees using the stack model. For the remaining data structures, run time was generally minimum when $m = 6$.

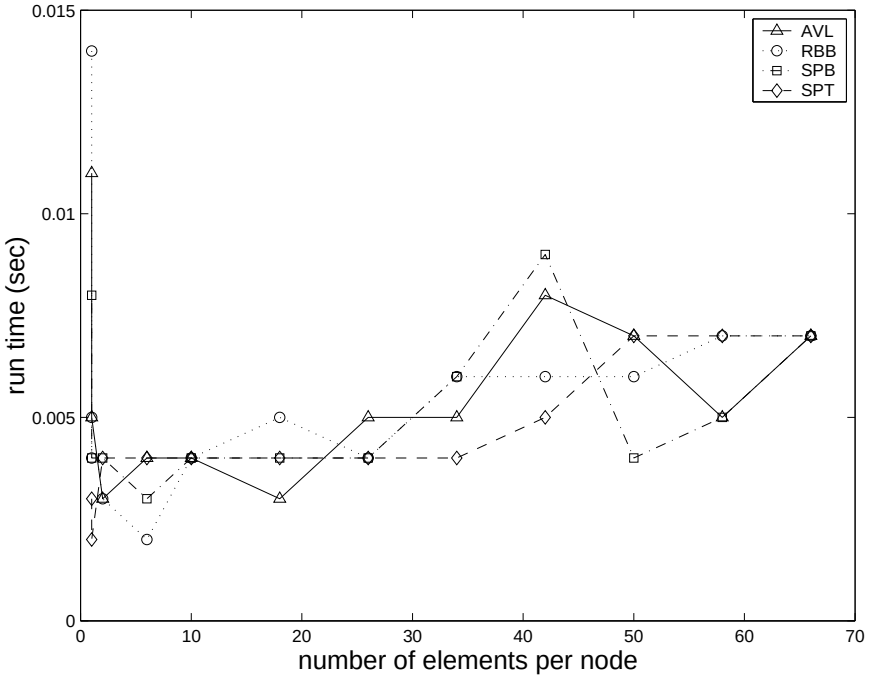


Figure 37: Run time in queue model: $n = 1$K

Histogramming Run Times

We programmed the histogramming application of dictionaries that is described in [5]. The input to the histogramming application is a sequence of n keys. However, only q% of these keys are distinct. In our experiments $q = 10$ (i.e., only 10% of the keys are distinct). The ouput is a list of the distinct keys (in ascending order) together with the frequency with which each key appears in the sequence. The problem is solved by performing n inserts into an initially empty dictionary tree (if the key being inserted is new, the key counter is set to one; if the insert key is already in the dictionary, the key counter is incremented). Following the n inserts, an inorder traversal of the tree is done to produce the desired frequency list in ascending order of key. During this inorder traversal, the unsorted elements (i.e., all elements other than the min and max elements) in a supernode are sorted into ascending order using insertion sort. For m larger than approximately 20 [2], the sort time could be reduced using quick sort in place of insertion sort. We did not do this for our experiments.

The time required to do the n inserts (n = 1M, 4M, and 16M) are shown in Figures 47 through 49. Figures 50 through 52 give the times for the inorder traversal plus the intranode insertion sort (the intranode sort is required only when $m > 3$).

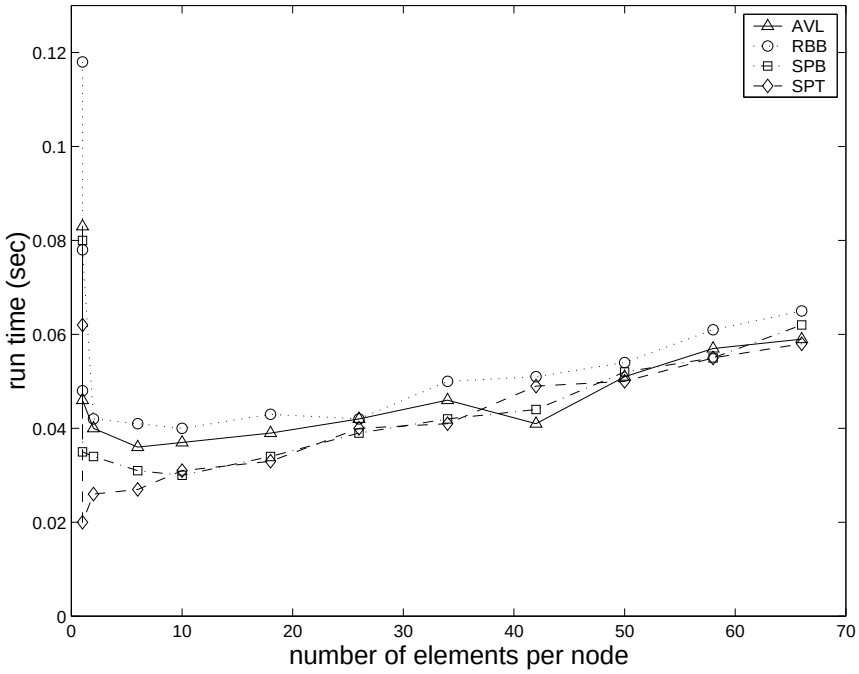


Figure 38: Run time in queue model: $n = 10\text{K}$

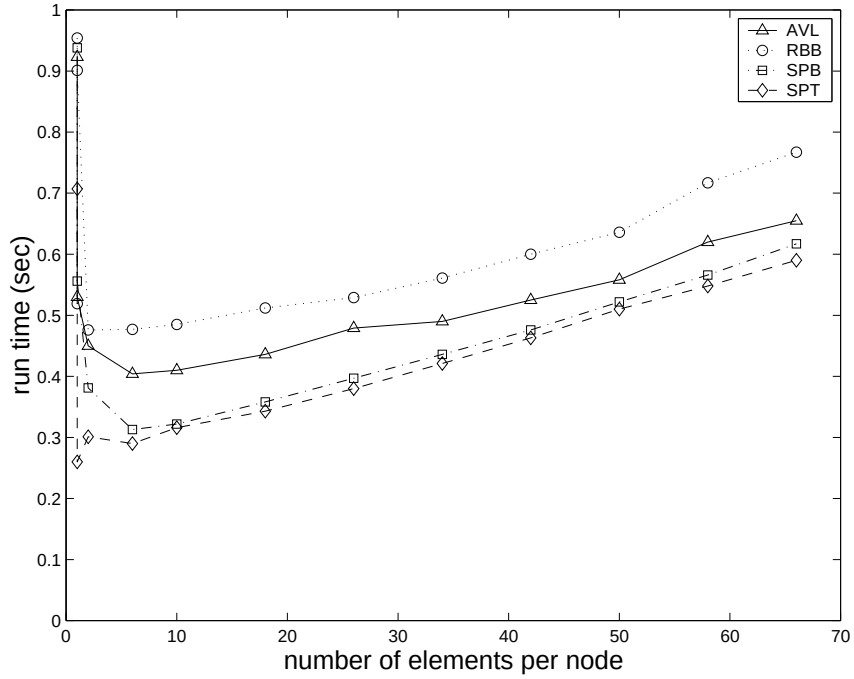


Figure 39: Run time in queue model: $n = 100\text{K}$

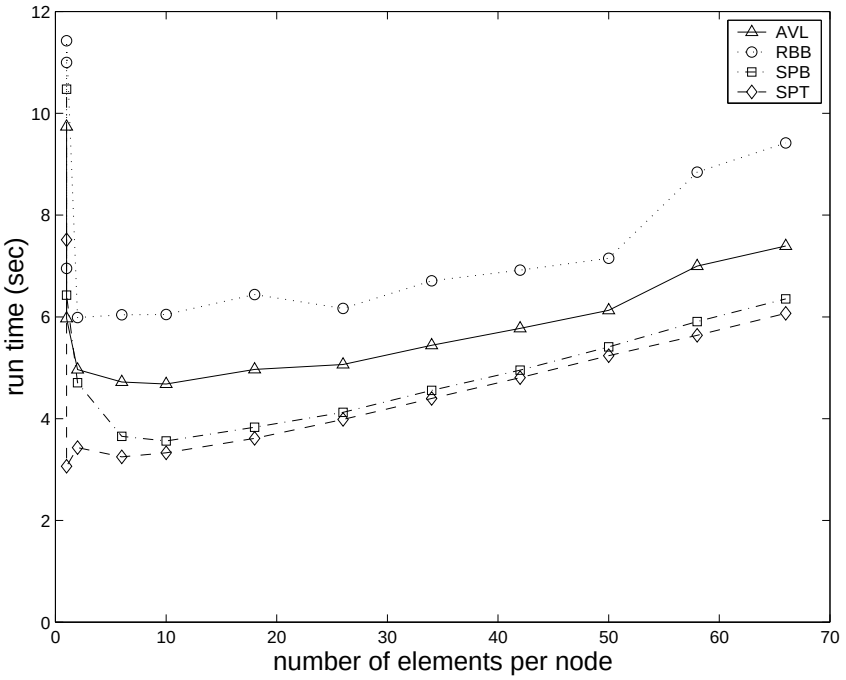


Figure 40: Run time in queue model: $n = 1\text{M}$

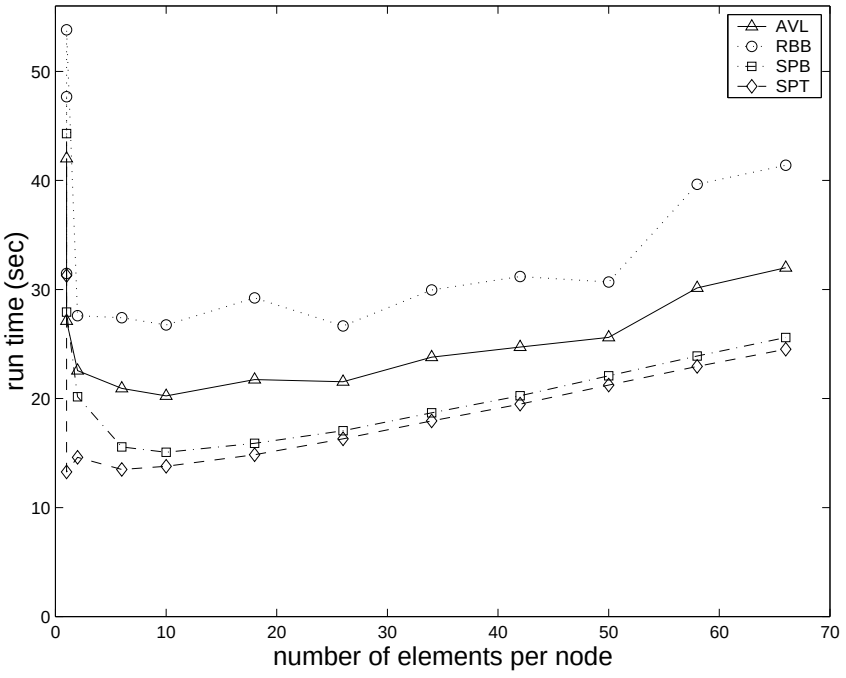


Figure 41: Run time in queue model: $n = 4\text{M}$

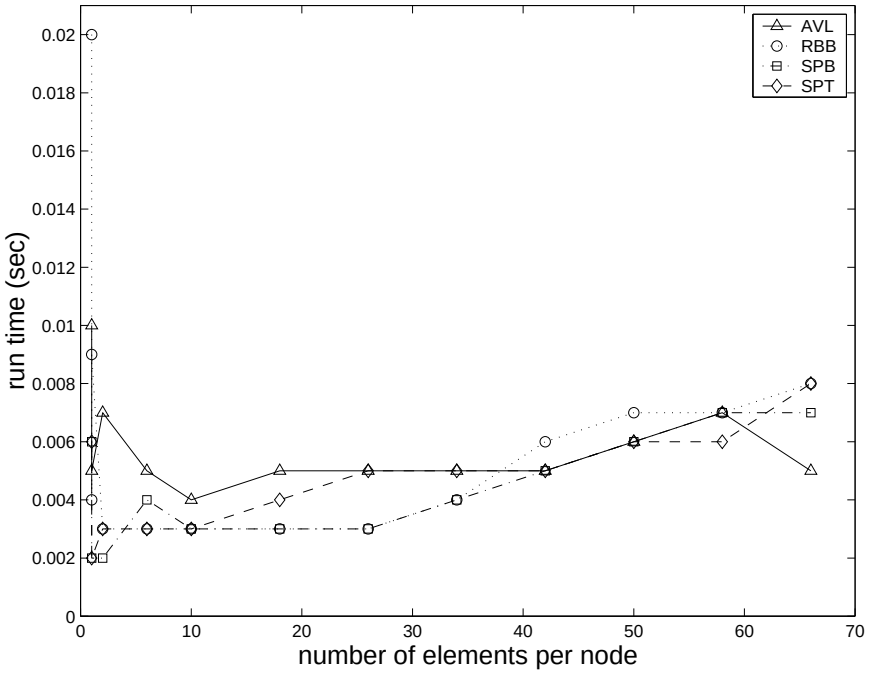


Figure 42: Run time in stack model: $n = 1\text{K}$

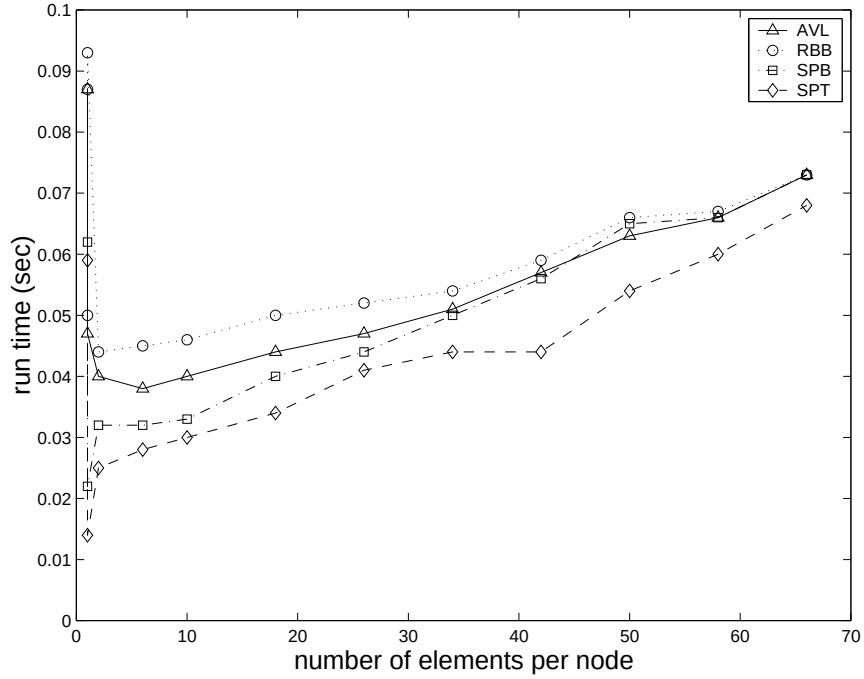


Figure 43: Run time in stack model: $n = 10\text{K}$

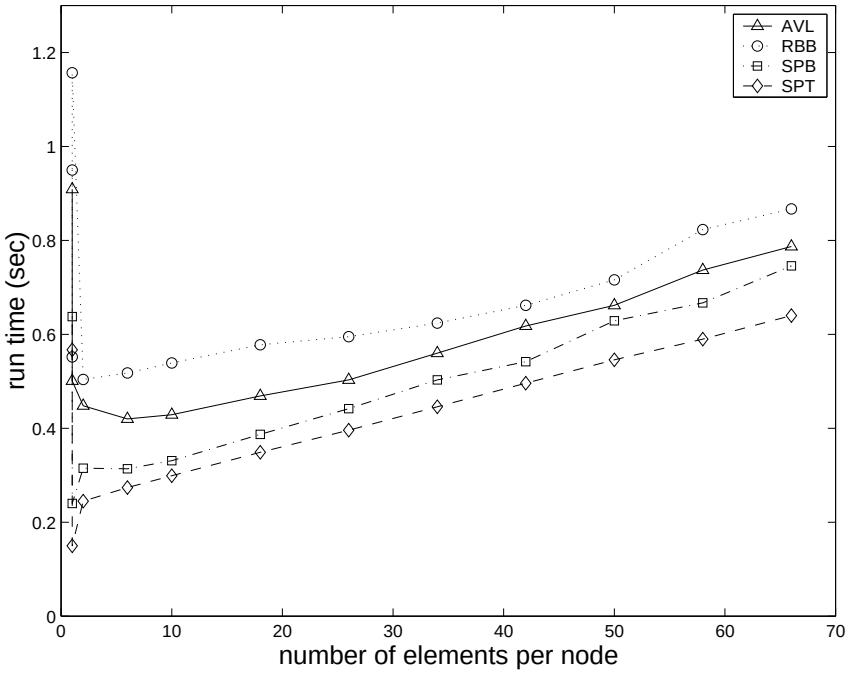


Figure 44: Run time in stack model: $n = 100\text{K}$

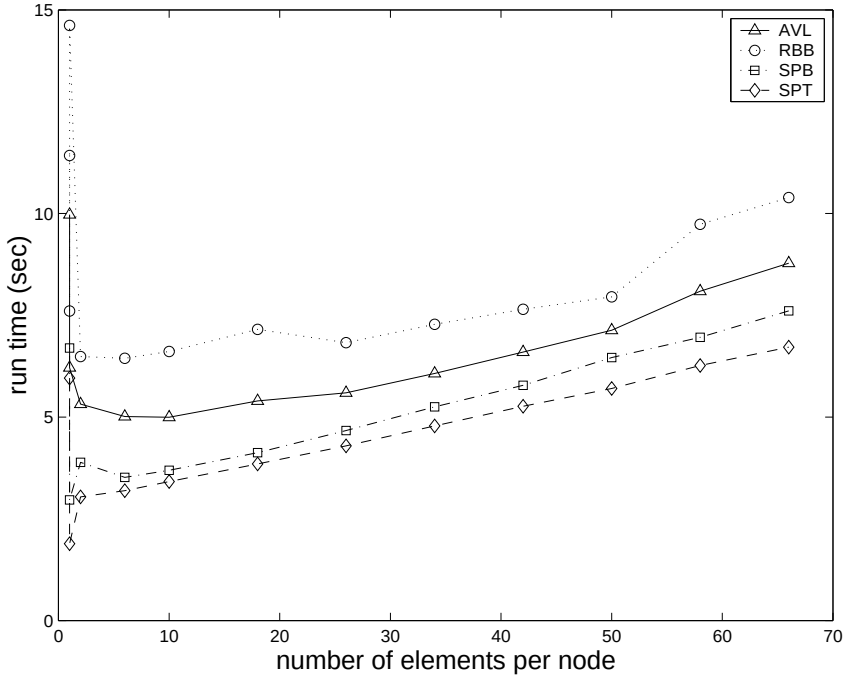


Figure 45: Run time in stack model: $n = 1\text{M}$

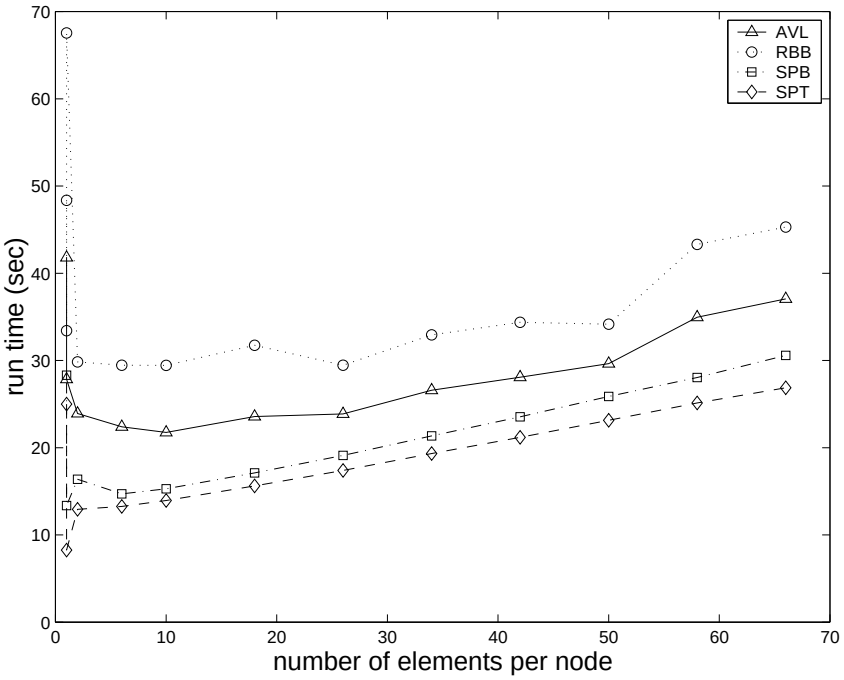


Figure 46: Run time in stack model: $n = 4$M

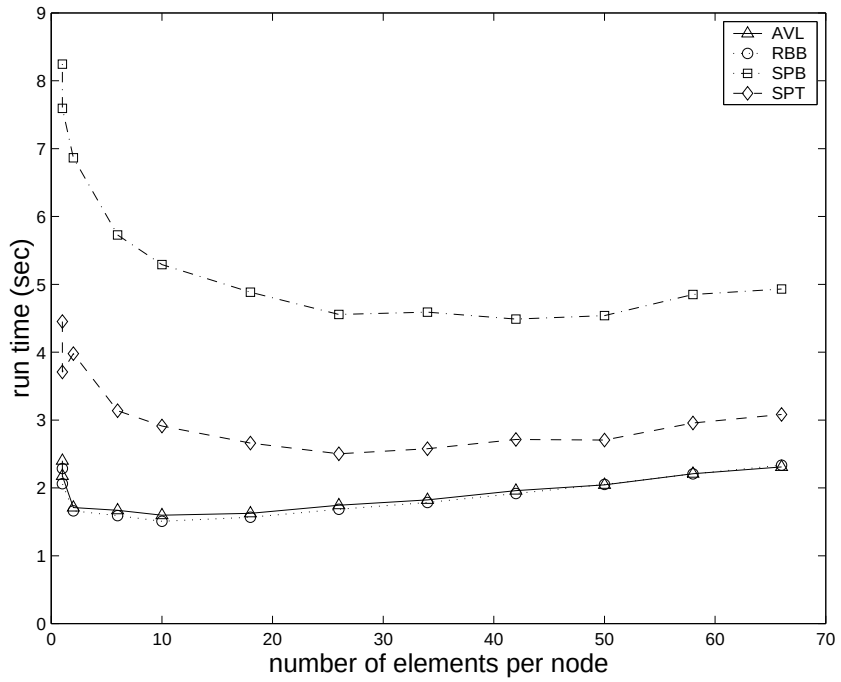


Figure 47: Run time for histogram inserts: $n = 1$M

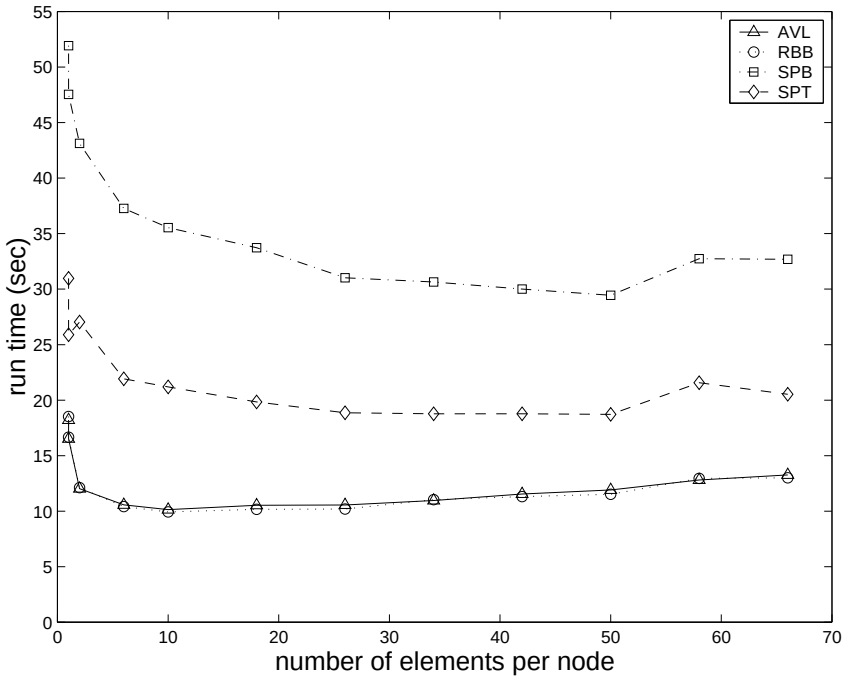


Figure 48: Run time for histogram inserts: $n = 4$M

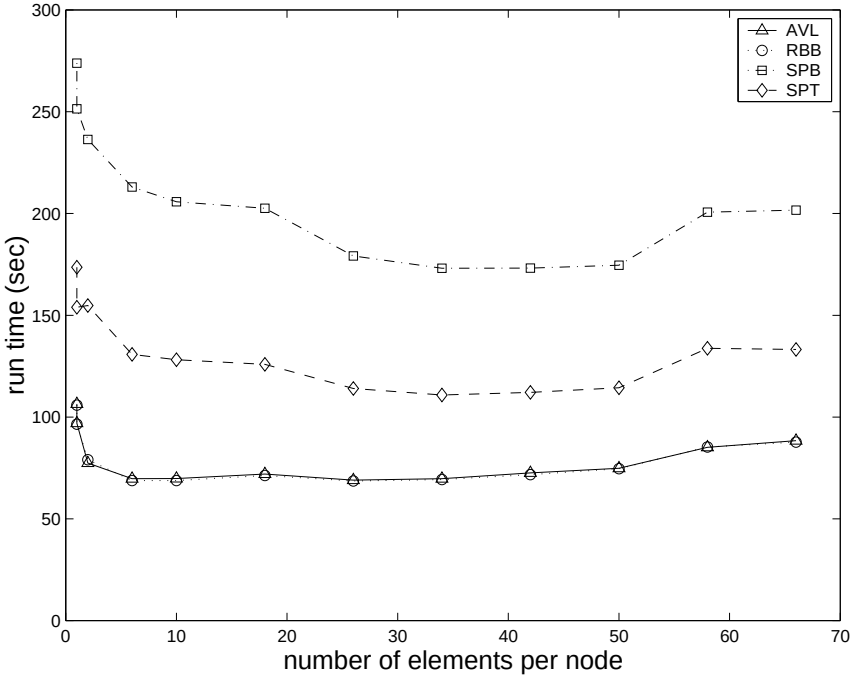


Figure 49: Run time for histogram inserts: $n = 16$M

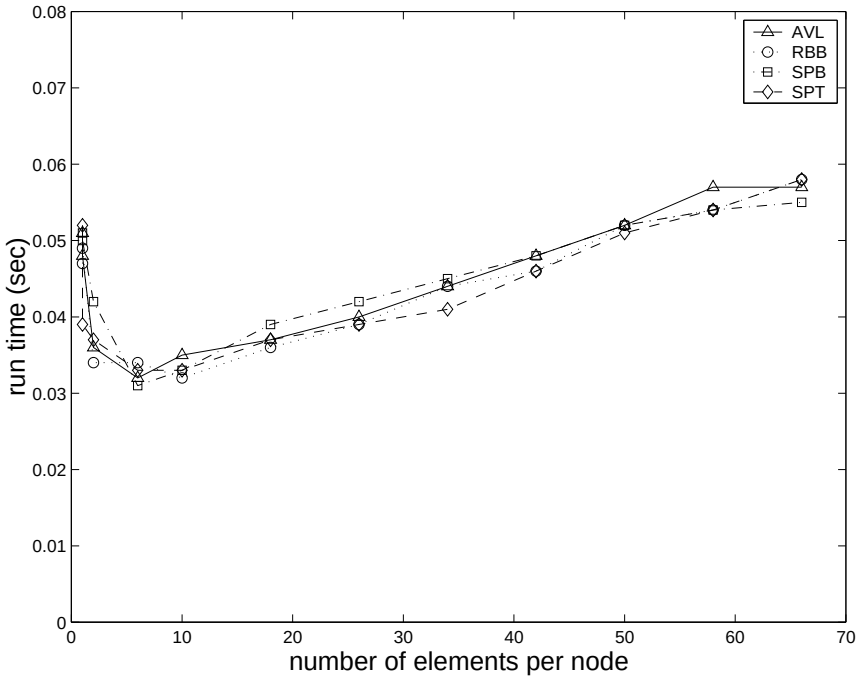


Figure 50: Run time for histogram output: $n = 1\text{M}$

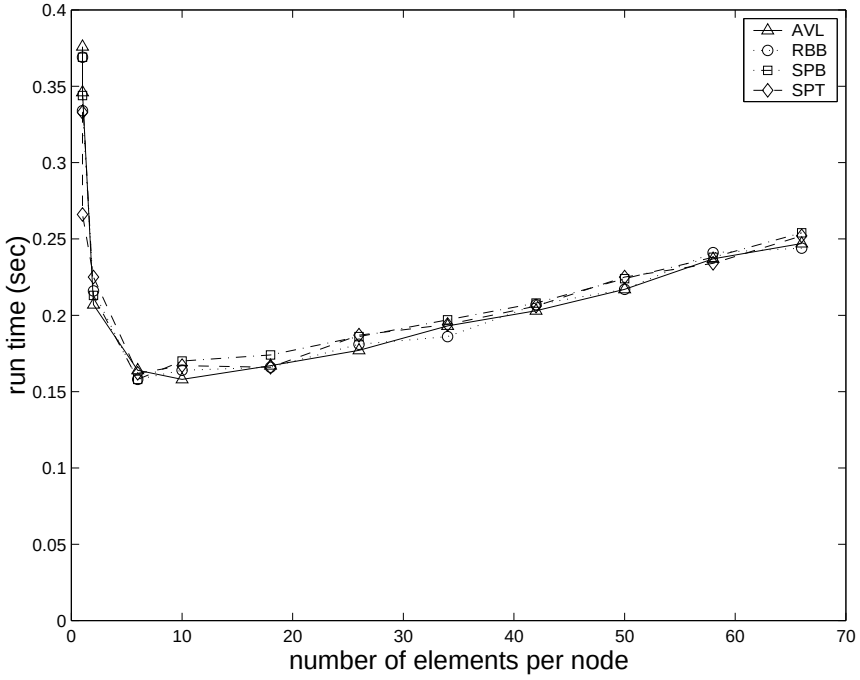


Figure 51: Run time for histogram output: $n = 4\text{M}$

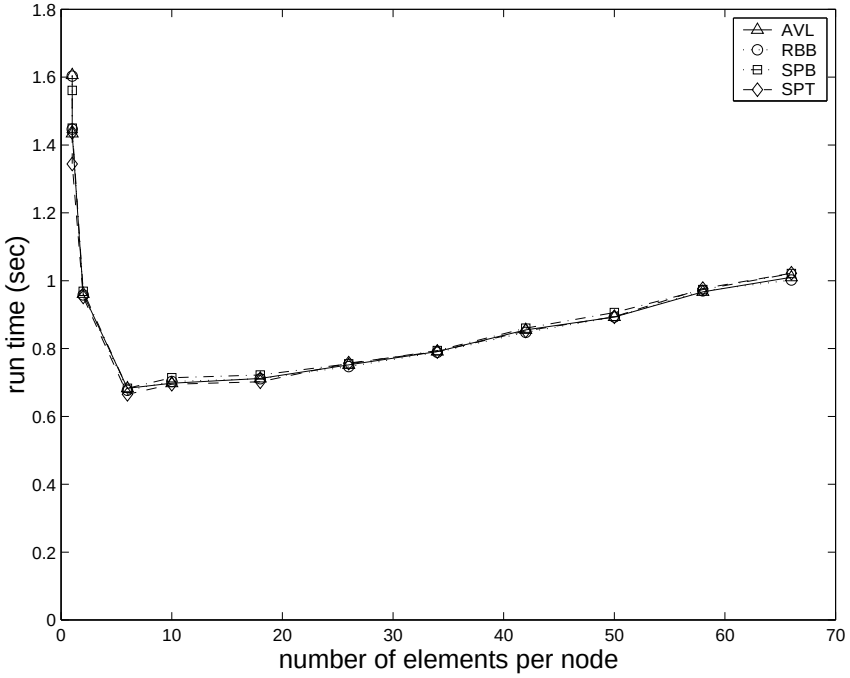


Figure 52: Run time for histogram output: $n = 16$M

For the histogram inserts, we make the following observations:

1. AVL and red-black trees are very competitive, both clearly outperform top down and bottom up splay trees, and top down splay trees provide superior performance relative to bottom up splay trees.

2. For AVL and red-black trees, run time is minimized when $m = 10$. However, run time is reasonably flat between $m = 10$ and $m = 50$. So, use of an m larger than 10 is recommended to reduce memory requirements.

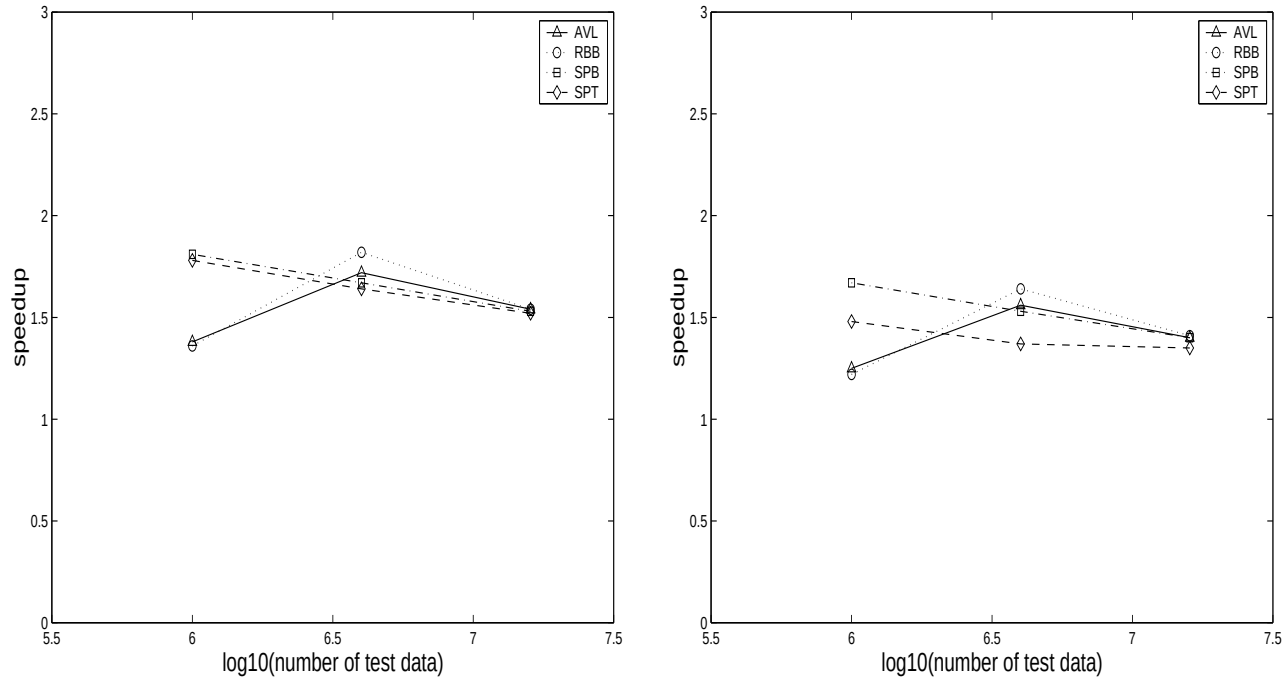
3. For splay trees, run time was minimum and reasonably flat between $m = 26$ and $m = 50$.

4. Figure 53 shows the speedup obtained by our supernode scheme when $m = 26$.

For the inorder traversal and sorts needed to generate the output frequency list, we make the following observations:

1. All four data structures are very competitive.

2. Even though no intranode sorts are done when $m = 1$, run time is minimized when supernodes with $m = 6$ (or 10) are used.



(a) relative to m=1 and non aligned

(b) relative to m=1 and aligned

Figure 53: Speedup in histograming when m = 26

8 Conclusion

We have proposed a supernode scheme for the implementation of dictionary structures such as AVL trees, red-black trees, and splay trees. Extensive experiments conducted by us show that the supernode scheme is a practical scheme that results in both memory and time reduction relative to the conventional single element per node implementation of these dictionary structures.

References

[1] R. Graham, D. Knuth, and O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, Reading, MA, 1990.

[2] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*, W. H. Freeman, San Francisco, 1995.

[3] D. Jones, "An empirical comparison of priority-queue and event-set implementation," *Communications of the Association for Computing Machinery*, 29(4), pp. 300–311, 1986.

[4] D. Knuth, *The Art of Computer Programming*, Volume 3, *Sorting and Searching*, Second Edition, Addison Wesley, New York, 1998.

[5] S. Sahni, *Data Structures, Algorithms, and Applications in Java*, McGraw Hill, New York, 2000.

[6] R. Sedgewick, *Algorithms in C++*, Addison-Wesley, New York, 1992.

[7] D. Sleator and R. Tarjan, "Self-adjusting binary search trees," *Journal of the Association for Computing Machinary*, 32(3), pp. 652–686, 1985.